

Decidable and Undecidable Fragments of Asynchronous Subtyping for Session Types

Mario Bravetti¹, Marco Carbone², and Gianluigi Zavattaro¹

- 1 Department of Computer Science, University of Bologna, Mura A. Zamboni 7, 40127 Bologna, Italy
`{mario.bravetti,gianluigi.zavattaro}@unibo.it`
- 2 Department of Computer Science, IT University of Copenhagen, Rued Langgaards Vej 7, 2300 Copenhagen, Denmark
`carbonem@itu.dk`

Abstract

Session types are behavioural types for guaranteeing that some programs are free from basic communication errors. Recent work has shown that the notion of asynchronous subtyping for session types is undecidable. However, it is not clear what the possible alternatives for making such relation decidable are. In this work, we propose two algorithms for deciding restricted but practically relevant definitions of asynchronous subtyping. Additionally, we further refine the existing undecidability results by showing how two restricted forms of asynchronous subtyping remain undecidable.

1998 ACM Subject Classification F.1.1, F.3.2, F.3.3

Keywords and phrases Session Types, Asynchronous Subtyping, Undecidability.

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction

Session types [4, 5] are types for controlling the communication behaviour of processes over channels. In a very simple but effective way, they express the pattern of sends and receives that a process must perform. Since they can guarantee freedom from some basic programming errors, session types are becoming popular with many main stream language implementations, e.g., Haskell [9], GO [11] or RUST [7].

Traditionally, types for programming languages make use of the notion of subtyping, a relation between types that allows to safely replace programs with other programs: a given program P with type S can always be safely replaced by another program Q with type T whenever $T \leq S$ (T is a subtype of S). In the literature, subtyping has been thoroughly studied for both binary session types (types expressing communication patterns between two entities) and multiparty session types [6] (types expressing communication patterns among many entities). In particular, whenever communication is synchronous, it is possible to compute whether two types are related by the subtyping relation [3]. Recently, Bravetti et al. [1] and Lange and Yoshida [8] have independently shown that subtyping becomes undecidable when communication is asynchronous, i.e., outputs are non-blocking. If $T \leq S$, type S must be able to simulate every input or output that T does. However, in the asynchronous setting, S is allowed to anticipate outputs that are prefixed by inputs, i.e., replacing a program of type S with one of type T that performs an output earlier is safe.

Although asynchronous subtyping is undecidable, it is still important to reason about cases for which the relation is decidable. This is because session types have become popular



© Mario Bravetti, Marco Carbone and Gianluigi Zavattaro;
licensed under Creative Commons License CC-BY

42nd Conference on Very Important Topics (CVIT 2016).

Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:29



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

in main stream programming languages, and, in such cases, asynchronous communications are the norm rather than the exception. The contribution of this work is two-fold: we show how minor changes to the subtyping relation or to the syntax of types make subtyping decidable; and, we refine the undecidability of subtyping to a more fine-grain relation.

Summary of our Main Contributions.

- *Concise Definition of Subtyping (§ 2).* We introduce a new, elegant, way of defining asynchronous subtyping where, in $T \leq S$, outputs in S cannot be indefinitely delayed (orphan message freedom). The new definition is based on just adding a constraint about closure under duality to the standard definition of asynchronous subtyping. We then show the new definition to be equivalent to that of Chen et al [2].
- *Decidability of k -bounded Subtyping (§ 3.2).* Our first decidability result focuses on a variant of subtyping, called k -bounded asynchronous subtyping. Our relation bounds the number of inputs that can prefix an output that must be anticipated. This is equivalent to putting an upper limit to a communication queue between two parties, a common fact in practice. We give and prove correct an algorithm for deciding whether any two session types are in a k -bounded subtyping relation.
- *Decidability of Single-Out and Single-In Session types (§ 3.3).* We consider the classes of single-out session types and single-in session types. In general, session types give the possibility of using branching for both inputs and outputs. In the case of inputs, this results in an external choice: a process exposes different labels that the sender of a message can pick. On the other hand, branching outputs implement internal choices, i.e., when a system decides to call a particular operation or another. Single-out session types are session types where outputs are always singleton (hence no internal choice). This sort of behaviour is very common in web-services where a server accepts alternative clients requests (input branching) but then it always reacts deterministically (concerning communication behaviour and not data). We give and prove correct an algorithm for deciding whether two single-out types are in the subtyping relation. Such an algorithm can thus be used, e.g., in typing systems for server code. Similarly, we consider single-in session types, where, as for clients in web-services, outputs are internally chosen and the corresponding inputs are always singletons. Then, exploiting a property of the subtyping relation (closure under duality), we show how to use the algorithm for deciding subtyping of single-out session types to decide subtyping of single-in session types. Such an algorithm can thus be used, e.g., in typing systems for client code.
- *Undecidability of Asynchronous Subtyping with Output Covariance and Input Contravariance (§ 4.2).* Subtyping for session types makes use of output covariance and input contravariance: an output $\oplus\{l_i : T_i\}_{i \in I}$ is a subtype of an output with more branches $\oplus\{l_j : T_j\}_{j \in J}$, for $I \subset J$; and an input $\&\{l_i : T_i\}_{i \in I}$ is a subtype of an input with less branches $\&\{l_j : T_j\}_{j \in J}$, for $J \subset I$. Existing results on the undecidability of asynchronous subtyping make use of such properties. We give an encoding of queue machines, a Turing-equivalent formalism, into subtyping which does not use output covariance and input contravariance, and show that, also with such restriction, subtyping remains undecidable.
- *Undecidability of Bounded Asynchronous Subtyping (§ 4.3).* We define bounded asynchronous subtyping as the union of the k -bounded subtyping relations, for all k . Such subtyping relates types that do not unboundedly put messages in a queue. We show that deciding whether there exists a k for which two types are in the k -bounded subtyping relation is undecidable.

Note. Detailed proofs of Theorems, Lemmas and Propositions can be found in the Appendix.

2 Session Types and Subtyping

We begin by formally introducing the various ingredients needed for our technical development. We give the formal syntax of a variant of binary session types, firstly introduced by Honda et al. [4]. Our variant merges outputs with internal choice and inputs with external choice:

► **Definition 1** (Session types). Given a set of labels L , ranged over by l , the syntax of binary session types is given by the following grammar:

$$T ::= \oplus\{l_i : T_i\}_{i \in I} \mid \&\{l_i : T_i\}_{i \in I} \mid \mu t.T \mid \mathbf{t} \mid \mathbf{end}$$

We assume guarded recursion by imposing $T \neq \mathbf{t}$ in $\mu t.T$. A session type is *single-out* if for all of its subterms $\oplus\{l_i : T_i\}_{i \in I}$ we have that $|I| = 1$. Similarly, a session type is *single-in* if for all of its subterms $\&\{l_i : T_i\}_{i \in I}$ we have that $|I| = 1$.

Above, the *output* type $\oplus\{l_i : T_i\}_{i \in I}$ denotes the type of a channel over which we can send one of the labels l_i (for $i \in I$). Dually, the *input* type $\&\{l_i : T_i\}_{i \in I}$ denotes the type of a channel ready to receive one of the labels l_i . Note that we abstract from the type of the message that could be sent over the channel, since this is orthogonal to our theory. Types $\mu t.T$ and \mathbf{t} denote standard tail recursion for recursive types. Type \mathbf{end} denotes the type of a channel that can no longer be used.

In our development, it is crucial to count the number of times we need to unfold a recursion $\mu t.T$. This is formalised by the following function:

► **Definition 2** (n -unfolding).

$$\begin{aligned} \text{unfold}^0(T) &= T & \text{unfold}^1(\oplus\{l_i : T_i\}_{i \in I}) &= \oplus\{l_i : \text{unfold}^1(T_i)\}_{i \in I} \\ \text{unfold}^1(\mu t.T) &= T\{\mu t.T/\mathbf{t}\} & \text{unfold}^1(\&\{l_i : T_i\}_{i \in I}) &= \&\{l_i : \text{unfold}^1(T_i)\}_{i \in I} \\ \text{unfold}^1(\mathbf{t}) &= \mathbf{t} & \text{unfold}^1(\mathbf{end}) &= \mathbf{end} & \text{unfold}^n(T) &= \text{unfold}^1(\text{unfold}^{n-1}(T)) \end{aligned}$$

Our definition of asynchronous subtyping uses the notion of input context, a type context consisting of a sequence of inputs preceding holes where types can be placed:

► **Definition 3** (Input Context). An input context \mathcal{A} is a session type with multiple holes defined by the syntax:

$$\mathcal{A} ::= []^n \mid \&\{l_i : \mathcal{A}_i\}_{i \in I}$$

An input context \mathcal{A} is well-formed whenever all its holes are consistently enumerated. Given an input context \mathcal{A} with holes indexed over $\{1, \dots, m\}$, we use $\mathcal{A}[T_k]^{k \in \{1, \dots, m\}}$ to denote the type obtained by filling each hole k in \mathcal{A} with the corresponding term T_k .

For session types, we define the usual notion of duality: given a session type T , its dual \bar{T} is defined as: $\overline{\oplus\{l_i : T_i\}_{i \in I}} = \&\{l_i : \bar{T}_i\}_{i \in I}$, $\overline{\&\{l_i : T_i\}_{i \in I}} = \oplus\{l_i : \bar{T}_i\}_{i \in I}$, $\overline{\mathbf{end}} = \mathbf{end}$, $\overline{\mathbf{t}} = \mathbf{t}$, and $\overline{\mu t.T} = \mu \bar{t}.\bar{T}$. In the sequel, we say that a relation \mathcal{R} on session types is *dual closed* if $(S, T) \in \mathcal{R}$ implies $(\bar{T}, \bar{S}) \in \mathcal{R}$. We are now ready to give the formal definition of the asynchronous subtyping relation.

► **Definition 4** (Asynchronous Subtyping, \leq). \mathcal{R} is an asynchronous subtyping relation whenever it is *dual closed* and $(T, S) \in \mathcal{R}$ implies that:

1. if $T = \mathbf{end}$ then $\exists n \geq 0$ such that $\text{unfold}^n(S) = \mathbf{end}$;
2. if $T = \oplus\{l_i : T_i\}_{i \in I}$ then $\exists n \geq 0, \mathcal{A}$ such that
 - $\text{unfold}^n(S) = \mathcal{A}[\oplus\{l_j : S_{kj}\}_{j \in J_k}]^{k \in \{1, \dots, m\}}$,

- $\forall k \in \{1, \dots, m\}. I \subseteq J_k$ and
- $\forall i \in I, (T_i, \mathcal{A}[S_{ki}]^{k \in \{1, \dots, m\}}) \in \mathcal{R}$;
- 3. if $T = \&\{l_i : T_i\}_{i \in I}$ then $\exists n \geq 0$ such that $\text{unfold}^n(S) = \&\{l_j : S_j\}_{j \in J}, J \subseteq I$ and $\forall j \in J. (T_j, S_j) \in \mathcal{R}$;
- 4. if $T = \mu t. T'$ then $(T'\{T/t\}, S) \in \mathcal{R}$.

T is an asynchronous subtype of S , written $T \leq S$, if there is an asynchronous subtyping relation \mathcal{R} such that $(T, S) \in \mathcal{R}$.

Intuitively, two types T and S are related by the relation \leq , whenever S is able to simulate T , but with a few twists: in the simulation game, type T is allowed to anticipate outputs nested in its syntax tree (asynchrony); and, output and input types enjoy covariance and contravariance, respectively.

We observe that our definition is formally different from the ones found in literature. In particular, it requires the subtyping relation to be dual close. In the sequel, we show that, with such a restriction, our subtyping is equivalent to the standard notion of asynchronous subtyping given by Dezani et al. [2] that is sensitive to orphan messages:

► **Definition 5** (Orphan-Message-Free Subtyping, \leq_o). \mathcal{R} is an orphan-message-free subtyping relation whenever $(T, S) \in \mathcal{R}$ implies 1., 3., and 4. in Definition 4, plus an extended version of 2. that contains also the following requirement:

- if $\mathcal{A} \neq []^1$ then $\forall i \in I. \& \in T_i$

where with $\& \in T_i$ we mean that T_i contains at least an external choice. T is a orphan-message-free subtype of S , written $T \leq_o S$, if there is a orphan-message-free subtyping relation \mathcal{R} such that $(T, S) \in \mathcal{R}$.

The definition above puts an extra restriction on our subtyping relation, namely it requires that if the smaller type contains no inputs, then there must be no input that could be delayed in the bigger type (no orphan messages [2]). Below, we show that such a restriction is not necessary since dual closeness guarantees orphan-message freedom:

► **Theorem 6.** *Given two session types S and T , we have $T \leq S$ if and only if $T \leq_o S$.*

3 Decidability Results

We now present decidability results for k -bounded asynchronous subtyping, a variant of asynchronous subtyping, and asynchronous subtyping for single-out/single-in session types.

3.1 A Subtyping Procedure

We start by giving a procedure for the general subtyping relation, which is known to be undecidable [1, 8]. In order to do so, we introduce two functions on the syntax of types. The function `outDepth` calculates how many unfolding are necessary for bringing an output outside a recursion. If that is not possible, the function is undefined (denoted by \perp).

► **Definition 7** (`outDepth`). The partial function `outDepth`(T, Γ), with Γ set of recursion variables, is inductively defined as:

$$\begin{aligned} \text{outDepth}(\oplus\{l_i : T_i\}_{i \in I}, \Gamma) &= 0 & \text{outDepth}(\&\{l_i : T_i\}_{i \in I}, \Gamma) &= \max\{\text{outDepth}(T_i, \Gamma) \mid i \in I\} \\ \text{outDepth}(\text{end}, \Gamma) &= \perp & \text{outDepth}(\mu t. T, \Gamma) &= \begin{cases} \perp & \text{if } t \in \Gamma \\ 1 + \text{outDepth}(T\{\mu t. T/t\}, \Gamma + \{t\}) & \text{otherwise} \end{cases} \end{aligned}$$

where $\max\{\text{outDepth}(T_i, \Gamma) \mid i \in I\} = \perp$, if $\text{outDepth}(T_i, \Gamma) = \perp$ for some $i \in I$; similarly, $1 + \perp = \perp$. We use `outDepth`(T) as a shorthand for `outDepth`(T, \emptyset).

For example, for any T_1 and T_2 , $\text{outDepth}(\oplus\{l_1 : T_1, l_2 : T_2\}) = 0$. On the other hand, consider the type $T_{\text{ex}} = \&\{l_1 : \mu\mathbf{t}. \oplus\{l_2 : T_1\}, l_3 : \mu\mathbf{t}. \&\{l_4 : \mu\mathbf{t}'. \oplus\{l_5 : T_2\}\}\}$: clearly, $\text{outDepth}(T_{\text{ex}}) = 2$. We then define $\text{outUnf}()$, a variant of the unfolding function given in Definition 2, which unfolds only where it is necessary, in order to reach an output:

► **Definition 8** (outUnf). The output unfolding $\text{outUnf}(T)$ is a partial function defined whenever $\text{outDepth}(T)$ is defined. Given $\text{outDepth}(T) = n$, $\text{outUnf}(T)$ is computed using the same inductive rules of $\text{unfold}^n(T)$, excluding the rule for $\oplus\{l_i : T_i\}_{i \in I}$ that, instead of recursively unfolding T_i , returns the same term $\oplus\{l_i : T_i\}_{i \in I}$.

The function above differs from unfold^n : for example, $\text{unfold}^2(T_{\text{ex}})$ would unfold twice both subterms $\mu\mathbf{t}. \oplus\{l_2 : T_1\}$ and $\mu\mathbf{t}. \&\{l_4 : \mu\mathbf{t}'. \oplus\{l_5 : T_2\}\}$. On the other hand, applying outDepth to the same term would unfold once the term in branch l_1 and twice the one in branch l_3 .

Subtyping Procedure. An environment Σ is a set containing pairs (T, S) , where T and S are types. We represent the states of our subtyping procedure as triples of the form $\Sigma \vdash T \leq_a S$ which intuitively read as “in order to succeed, the procedure must check whether T is a subtype of S , provided that pairs in Σ have already been visited”. Then, our *subtyping procedure* is defined as a transition relation $\Sigma \vdash T \leq_a S \rightarrow \Sigma' \vdash T' \leq_a S'$, such that if $\Sigma \vdash T \leq_a S$ matches the conclusions of one of the rules in Figure 1, then $\Sigma' \vdash T' \leq_a S'$ is produced by the corresponding premises. We give highest priority to rule **Asmp**, thus ensuring that at most one rule is applicable.¹ The main idea behind the environment Σ is to avoid cycles when dealing with recursive types. Rules **RecR₁** and **RecR₂** deal with the case in which the type on the right-hand side is a recursion and must be unfolded. If the type on the left-hand side is not an output then the procedure simply adds the current pair to Σ and continues. On the other hand, if an output must be found, we apply **RecR₁** which checks whether such output is available. Rule **Out** allows nested outputs to be anticipated (when not under recursion) and condition $(\mathcal{A} \neq []^1) \Rightarrow \forall i \in I. \& \in T_i$ makes sure there are no orphan messages. The remaining rules are self-explanatory. $\Sigma \vdash T \leq_a S \rightarrow^* \Sigma' \vdash T' \leq_a S'$ is the reflexive and transitive closure of such relation. We write $\Sigma \vdash T \leq_a S \rightarrow_{\text{err}}$ to mean that no rule can be applied to $\Sigma \vdash T \leq_a S$. Checking whether a type T is a subtype of a type S may be done by applying our procedure to $\emptyset \vdash T \leq_a S$.

► **Example 9.** Consider $T = \mu\mathbf{t}. \oplus\{l_1 : \&\{l_2 : \mathbf{t}\}\}$ and $S = \mu\mathbf{t}. \oplus\{l_1 : \&\{l_2 : \&\{l_2 : \mathbf{t}\}\}\}$. Clearly, the two types T and S are related by asynchronous subtyping, i.e. $T \leq S$. However, our subtyping procedure on $\emptyset \vdash T \leq_a S$ will not terminate: the right-hand side will grow indefinitely since every unfolding adds two inputs.

However, two types are not related by subtyping if and only if the subtyping procedure terminates with an error. That is, if $T \leq S$ then the procedure on $\emptyset \vdash T \leq_a S$ terminates successfully or diverges; instead, if $T \not\leq S$ then the procedure terminates with an error. This is shown by the following

► **Proposition 10.** *Given the types T and S , we have that there exist Σ', T', S' such that $\emptyset \vdash T \leq_a S \rightarrow^* \Sigma' \vdash T' \leq_a S' \rightarrow_{\text{err}}$ if and only if $T \not\leq S$.*

¹ The priority of **Asmp** is sufficient because all the other rules are alternative, i.e., given a judgement $\Sigma \vdash T \leq_a S$ there are no two rules different from **Asmp** that can be both applied.

$$\begin{array}{c}
(\mathcal{A} \neq []^1) \Rightarrow \forall i \in I. \& \in T_i \\
\frac{\forall n. I \subseteq J_n \quad \forall i \in I. \Sigma \vdash T_i \leq_a \mathcal{A}[S_{ni}]^n}{\Sigma \vdash \oplus\{l_i : T_i\}_{i \in I} \leq_a \mathcal{A}[\oplus\{l_j : S_{nj}\}_{j \in J_n}]^n} \text{Out} \quad \frac{J \subseteq I \quad \forall j \in J. \Sigma \vdash T_j \leq_a S_j}{\Sigma \vdash \&\{l_i : T_i\}_{i \in I} \leq_a \&\{l_j : S_j\}_{j \in J}} \text{In} \\
\\
\frac{}{\Sigma, (T, S) \vdash T \leq_a S} \text{Asmp} \quad \frac{}{\Sigma \vdash \text{end} \leq_a \text{end}} \text{End} \quad \frac{\Sigma, (\mu\mathbf{t}.T, S) \vdash T\{\mu\mathbf{t}.T/\mathbf{t}\} \leq_a S}{\Sigma \vdash \mu\mathbf{t}.T \leq_a S} \text{RecL} \\
\\
\frac{T = \text{end} \vee T = \&\{l_i : T_i\}_{i \in I} \quad \Sigma, (T, \mu\mathbf{t}.S) \vdash T \leq_a S\{\mu\mathbf{t}.S/\mathbf{t}\}}{\Sigma \vdash T \leq_a \mu\mathbf{t}.S} \text{RecR}_1 \\
\\
\frac{\text{outDepth}(S) \geq 1 \quad \Sigma, (\oplus\{l_i : T_i\}_{i \in I}, S) \vdash \oplus\{l_i : T_i\}_{i \in I} \leq_a \text{outUnf}(S)}{\Sigma \vdash \oplus\{l_i : T_i\}_{i \in I} \leq_a S} \text{RecR}_2
\end{array}$$

■ **Figure 1** A Procedure for Checking Subtyping

3.2 k -bounded Asynchronous Subtyping

Although asynchronous subtyping is undecidable, we can make it decidable by putting an upper-bound limit on the number of outputs that can be anticipated. We say that an input context \mathcal{A} is k -bounded if the maximal number of nested inputs in \mathcal{A} is less or equal to k .

► **Definition 11** (k -bounded Asynchronous Subtyping). The k -bounded asynchronous subtyping \leq^k is defined as in Definition 5, with the only difference that the input context \mathcal{A} in item 2. is assumed to be k -bounded.

We can then define an algorithm for k -bounded asynchronous subtyping building on the subtyping procedure defined previously. We define \leq_a^k as \leq_a with the only difference that the input context \mathcal{A} in rule Out in Figure 1, is assumed to be k -bounded. Then, the following result holds:

► **Theorem 12.** *The algorithm for \leq_a^k always terminates and, given the types T and S , there exist Σ', T', S' such that $\emptyset \vdash T \leq_a^k S \rightarrow^* \Sigma' \vdash T' \leq_a S' \rightarrow_{\text{err}}$ if and only if $T \not\leq_k S$.*

3.3 Asynchronous Subtyping for Single-Out and Single-In Types

We now move to another decidability result where, instead of restricting the definition of asynchronous subtyping, we restrict the set of types over which the relation is used. In our development, we focus on single-out types, session types where outputs are always singletons. In order to present our algorithm that determines the decidability of asynchronous subtyping for single-out types, we define the notions of leaf set, output anticipation and reachable types.

► **Definition 13** (Leaf Set). Given a session type S , we write $\text{noIn}(S)$ if S is not of the form $\&\{l_i : S_i\}_{i \in I}$. Given a session type T , we define

$$\text{leafSet}(T) = \{T_1, \dots, T_n \mid \text{noIn}(T_i) \text{ and } \exists \text{ input context } \mathcal{A} \text{ s.t. } T = \mathcal{A}[T_k]^{k \in \{1 \dots n\}}\}$$

The leaf set of a session type T is the set of subterms reachable from its root through a path of inputs. For example, the leaf set of the term $\&\{l_1 : \mu\mathbf{t}. \oplus\{l_2 : \mathbf{t}\}, l_3 : \&\{l_4 : \oplus\{l_2 : \mu\mathbf{t}. \oplus\{l_2 : \mathbf{t}\}\}\}\}$ is $\{\mu\mathbf{t}. \oplus\{l_2 : \mathbf{t}\}, \oplus\{l_2 : \mu\mathbf{t}. \oplus\{l_2 : \mathbf{t}\}\}\}$.

► **Definition 14** (Output Anticipation). The partial function $\text{antOut}(T, l_{i_1} \cdots l_{i_n})$, with T single-out session type and $l_{i_1} \cdots l_{i_n}$ sequence of labels, is inductively defined as follows:

$$\text{antOut}(T, l_{i_1} \cdots l_{i_n}) = \begin{cases} T & \text{if } n = 0 \\ \mathcal{A}[T_k]^k & \text{if } \text{outUnf}(\text{antOut}(T, l_{i_1} \cdots l_{i_{n-1}})) = \mathcal{A}[\oplus\{l_{i_n} : T_k\}]^k \end{cases}$$

We say that T can infinitely anticipate outputs, written $\text{antOutInf}(T)$, if there exists an infinite sequence of labels $l_{i_1} \cdots l_{i_j} \cdots$ such that $\text{antOut}(T, l_{i_1} \cdots l_{i_n})$ is defined for every n .

The function $\text{antOut}(T, \tilde{l})$ anticipates all outputs in the sequence \tilde{l} . For example, the function applied to $\&\{l : \mu\mathbf{t}. \oplus\{l_1 : \oplus\{l_2 : \mathbf{t}\}\}, l' : \oplus\{l_1 : \mu\mathbf{t}. \oplus\{l_2 : \oplus\{l_1 : \mathbf{t}\}\}\}$ and the sequence (l_1, l_2) would return the same term, while it would be undefined with the sequence (l_1, l_1) .

The definition of $\text{antOutInf}(T)$ is not algorithmic in that it quantifies on every possible natural number n . Nevertheless, it can be decided by checking whether for every session type obtained from T by means of output anticipations, all the terms populating its leaf set can anticipate the same output label. Despite such process possibly generates infinitely many session types, the terms populating the leaf sets are finite and are over-approximated by the function $\text{reach}(T)$, which always returns a finite set and is defined as:

► **Definition 15** (Reachable types). Given a single-out session type T , $\text{reach}(T)$ is the minimal set of session types such that:

1. $T \in \text{reach}(T)$;
2. $\&\{l_i : T_i\}_{i \in I} \in \text{reach}(T)$ implies $T_i \in \text{reach}(T)$ for every $i \in I$;
3. $\mu\mathbf{t}.T' \in \text{reach}(T)$ implies $T'\{\mu\mathbf{t}.T'/\mathbf{t}\} \in \text{reach}(T)$;
4. $\oplus\{l : T'\} \in \text{reach}(T)$ implies $T' \in \text{reach}(T)$.

Notice that $\text{reach}(T)$ is populated by those session types obtained by consuming in sequence the initial inputs and outputs, and by unfolding recursion only when it is at the top level. These terms are finite in that eventually the final term **end**, or a term already considered, are reached. The latter occurs after consumption of all the inputs and outputs in front of a recursion variable already unfolded.

► **Proposition 16.** *Given a single-out session type T , $\text{reach}(T)$ is finite and it is decidable whether $\text{antOutInf}(T)$.*

Subtyping algorithm for single-out types. Our algorithm is parametric on the session type Z , which is the type on the right-hand side of the initial pair of types to be checked (the algorithm is intended to check $V \leq Z$, for some type Z). We start from the initial judgement $\emptyset \vdash V \leq_{\mathbf{t}} Z$ and then apply from bottom to top the rules in Figure 1, where $\leq_{\mathbf{a}}$ is replaced by $\leq_{\mathbf{t}}$, plus the following additional rule:

$$\frac{S \in \text{reach}(Z) \quad \text{antOutInf}(S) \quad |\gamma| < |\beta| \quad \text{leafSet}(\text{antOut}(S, \gamma)) = \text{leafSet}(\text{antOut}(S, \beta))}{\Sigma, (T, \text{antOut}(S, \gamma)) \vdash T \leq_{\mathbf{t}} \text{antOut}(S, \beta)} \text{Asmp2}$$

This rule guarantees termination of the algorithm by catching all those cases where the term on the right grows indefinitely, by anticipating outputs and accumulating inputs. These infinitely many distinct types are anyway obtainable starting from the finite set $\text{reach}(Z)$, by means of output anticipations. Hence there exists $S \in \text{reach}(Z)$ that can generate infinitely many of these types: this guarantees $\text{antOutInf}(S)$ to be true. As observed above, the leafs of such infinitely many terms are themselves taken from the finite set $\text{reach}(Z)$. This guarantees that the algorithm, among the types that can be obtained from S , visits two terms having the same leaf set. These, even if syntactically different, are equivalent as far as the subtyping

game is concerned. In order to avoid the possibility of applying two distinct rules to the same judgement, we give rule **Asmp2** the same highest priority as rule **Asmp**. Also in this case, we use $\Sigma \vdash T \leq_t S \rightarrow \Sigma' \vdash T' \leq_t S'$ to denote that the latter can be obtained from the former by one rule application, and $\Sigma \vdash T \leq_t S \rightarrow_{\text{err}}$, to denote that there is no rule that can be applied to the judgement $\Sigma \vdash T \leq_t S$.

We can now state the termination and soundness of the algorithm:

► **Theorem 17.** *Given two single-out session types T and S , the algorithm applied to the initial judgement $\emptyset \vdash T \leq_t S$ terminates.*

► **Theorem 18.** *Given two single-out session types T and S , we have that there exist Σ', T', S' such that $\emptyset \vdash T \leq_a S \rightarrow^* \Sigma' \vdash T' \leq_a S' \rightarrow_{\text{err}}$ if and only if there exist Σ'', T'', S'' such that $\emptyset \vdash T \leq_t S \rightarrow^* \Sigma'' \vdash T'' \leq_t S'' \rightarrow_{\text{err}}$.*

Finally, we can conclude the decidability of asynchronous subtyping for single-out session types. Moreover, using dual closedness, the same result holds for single-in session types.

► **Corollary 19 (Decidability).** *Asynchronous subtyping for single-out and single-in session types is decidable.*

4 Undecidability Results

We now augment focus on the undecidability of asynchronous subtyping. Existing results [1, 8] show that asynchronous subtyping is undecidable. They encode termination of a Turing equivalent formalism into the subtyping game by exploiting covariance of outputs and contravariance of inputs. In this section we show that, even without covariance and contravariance, subtyping remains undecidable. Moreover, we show that the union, for all k , of all k -bounded subtyping relations (relating types that do not unboundedly put messages in a queue) is undecidable.

4.1 Queue Machines

We prove undecidability of asynchronous subtyping without using output covariance and input contravariance, by reduction from the acceptance problem for queue machines. Queue machines are a formalism similar to pushdown automata, but with a queue instead of a stack. Queue machines are Turing-equivalent.

► **Definition 20 (Queue machine).** A queue machine M is defined by a six-tuple $(Q, \Sigma, \Gamma, \$, s, \delta)$ where:

- Q is a finite set of states;
- $\Sigma \subset \Gamma$ is a finite set denoting the input alphabet;
- Γ is a finite set denoting the queue alphabet;
- $\$ \in \Gamma - \Sigma$ is the initial queue symbol;
- $s \in Q$ is the start state;
- $\delta : Q \times \Gamma \rightarrow Q \times \Gamma^*$ is the transition function.

A *configuration* of a queue machine is an ordered pair (q, γ) where $q \in Q$ is its current state and $\gamma \in \Gamma^*$ is the content of the queue (Γ^* is the Kleene closure of Γ). The starting configuration on an input string $x \in \Sigma^*$ is $(s, x\$)$. The transition relation \rightarrow_M from one configuration to the next one is defined as $(p, A\alpha) \rightarrow_M (q, \alpha\gamma)$, when $\delta(p, A) = (q, \gamma)$. A machine M accepts an input x if it blocks by emptying the queue. Formally, x is accepted

by M if $(s, x\$) \rightarrow_M^* (q, \epsilon)$ where ϵ is the empty string and \rightarrow_M^* is the reflexive and transitive closure of \rightarrow_M . Intuitively, a queue machines is a Turing machine with a special tape that works as a FIFO queue. Below, we define a subclass of queue machines, called single consuming queue machines:

► **Definition 21** (Single Consuming Queue machine). We say that a queue machine $M = (Q, \Sigma, \Gamma, \$, s, \delta)$ is *single consuming* if $\delta(q, a) = (q', \epsilon)$, for some q, a and q' , implies that there exist no b and q'' such that $\delta(q', b) = (q'', \epsilon)$.

Single consuming queue machines are Turing-equivalent:

► **Theorem 22.** *Given a single consuming queue machine M and an input x , the termination of M on x is undecidable.*

4.2 Undecidability of Asynchronous Subtyping without Output Covariance and Input Contravariance

We prove the undecidability of asynchronous subtyping without output covariance and input contravariance by encoding single consuming queue machines into subtyping. A queue machine has two main components, the finite control and the queue. Our encoding generates a pair of types, say T and S , such that T encodes the finite control and S encodes the queue. Then, the subtyping $T \leq S$ simulates the execution of the machine. We now give the formal definition of our encoding and then give a detailed explanation.

In the sequel, our encoding associates each state q of a queue machine with a unique recursion variable \mathbf{q} . Symbols of the alphabet Γ are used as labels by the session types generated by the encoding. Finally, we define \uplus as $\{l_i : T_i\}_{i \in I} \uplus \{l_j : T_j\}_{j \in J} = \{l_k : T_k\}_{k \in I \cup J}$.

► **Definition 23** (Encoding). Let $M = (Q, \Sigma, \Gamma, \$, s, \delta)$ be a queue machine such that $q \in Q$, $S \subseteq Q$ and $C_1, \dots, C_m \in \Gamma$, with $m \geq 0$. Then, the *finite control encoding function* $\llbracket q \rrbracket^S$ and the *queue encoding function* $\llbracket C_1 \dots C_m \rrbracket$ are defined as in Figure 2(a) and Figure 2(b) respectively.

The encoding generates two types T and S such that T encodes the transition function of the encoded queue machine (finite control), while S encodes the queue. The main idea is for type T to be able to perform an input on each of the symbols in Γ , and continue according to the definition of the transition function δ . Type S then matches such input with the correct symbol depending on the state of the queue. For example, if $\Gamma = \{A, B\}$ and symbol A is on the head of the queue, we have $T = \&\{A : \dots, B : \dots\}$ and $S = \&\{A : \dots\}$: type T is able to react to any symbol that may be present on the queue (like the transition function δ), while type S reacts with the actual value on the queue, symbol A . Unfortunately, such idea exploits contravariance for inputs. Therefore, it must be the case that the input in S is of the form $\&\{A : \dots, B : \dots\}$. Our encoding makes sure that if label A is selected then the simulation of the queue machine continues. Otherwise, an infinite simulation is started (starting from B in the example). Insertion of symbols is simulated by performing outputs. Such outputs are matched by the queue which also releases a corresponding input. Also in this case, we have a problem without covariance of outputs which forces us to introduce extra paths in the subtyping game. This makes the game highly non-deterministic and such that several paths that the game can take differ from what the encoded machine does. We discuss in detail the various cases which our encoding in Figure 2 can be in:

1. *The encoding of the finite control reads the correct symbol.* We represent the machine reading a symbol A from the queue while being in state q , with an input type of the

a) *Finite Control*

$$\llbracket q \rrbracket^S = \begin{cases} \mu \mathbf{q} . \& \{ A : \llbracket B_1^A \dots B_{n_A}^A \rrbracket_{q'}^{S \cup \{q\}} \}_{A \in \Gamma} & \text{if } q \notin S \text{ and } \delta(q, A) = (q', B_1^A \dots B_{n_A}^A) \\ \mathbf{q} & \text{if } q \in S \end{cases}$$

b) *Queue*

$$\llbracket C_1 \dots C_m \rrbracket = \begin{cases} \mu \mathbf{t} \oplus \{ A : \& (\{ A : \mathbf{t} \} \uplus \{ A' : T'' \}_{A' \in \Gamma \setminus \{A\}}) \}_{A \in \Gamma} & \text{if } m = 0 \\ \& (\{ C_1 : \llbracket C_2 \dots C_m \rrbracket \} \uplus \{ A' : T'' \}_{A' \in \Gamma \setminus \{C_1\}}) & \text{otherwise} \end{cases}$$

where $\llbracket B_1 \dots B_m \rrbracket_r^\tau = \begin{cases} \llbracket r \rrbracket^\tau & \text{if } m = 0 \\ \oplus (\{ B_1 : \llbracket B_2 \dots B_m \rrbracket_r^\tau \} \uplus \{ A' : T' \}_{A' \in \Gamma \setminus \{B_1\}}) & \text{otherwise} \end{cases}$

$$T' = \mu \mathbf{t} . \& \{ A_1 : \oplus \{ A_2 : \mathbf{t} \}_{A_2 \in \Gamma} \}_{A_1 \in \Gamma}$$

$$T'' = \mu \mathbf{t} . \& \{ A_1 : \& \{ A_2 : \oplus \{ A_3 : \mathbf{t} \}_{A_3 \in \Gamma} \}_{A_2 \in \Gamma} \}_{A_1 \in \Gamma}$$

■ **Figure 2** Encoding of the Finite Control and the Queue of a Queue Machine

form $\& \{ A : \llbracket B_1^A \dots B_{n_A}^A \rrbracket_{q'}^{S \cup \{q\}} \}_{A \in \Gamma}$, where each branch corresponds to a possible symbol that can be read. On the other hand, a queue $C_1 \dots C_m$ is encoded as an input type of the form $\& (\{ C_1 : \llbracket C_2 \dots C_m \rrbracket \} \uplus \{ A' : T'' \}_{A' \in \Gamma \setminus \{C_1\}})$ where the branch with label C_1 represents the actual content of the queue. Hence, in the simulation game, if the finite control reads symbol A and this is matched by the correct symbol in the queue, then the type $\llbracket B_1^A \dots B_{n_A}^A \rrbracket_{q'}^{S \cup \{q\}}$ deals with inserting symbols $B_1^A \dots B_{n_A}^A$ into the queue.

2. *The encoding of the finite control reads the wrong symbol.* In this case, the encoding of the finite control picks a symbol that is not the symbol in the head of the queue. In order to match such action, the encoding of the queue will take the branch $\{ A' : T'' \}_{A' \in \Gamma \setminus \{C_1\}}$. From now on, T'' is designed in a way that it can match every move the finite control can do, by repeatedly alternating two inputs with a subsequent output on every queue symbol. Note that, since inputs cannot be anticipated matching every move is only feasible if the encoded queue machine is single consuming.
3. *The encoding of the finite control writes the correct symbol.* Once the finite control has read a symbol, it performs $\llbracket B_1 \dots B_m \rrbracket_r^\tau$, whose objective is to simulate the writing of $B_1 \dots B_m$ into the queue. If such string is empty then it moves to the encoding of the next state according to transition function δ . Otherwise, it translates to the type $\oplus (\{ B_1 : \llbracket B_2 \dots B_m \rrbracket_r^\tau \} \uplus \{ A' : T' \}_{A' \in \Gamma \setminus \{B_1\}})$. The queue, in order to match B_1 (and then B_2, \dots, B_m) can always anticipate outputs with the term $\mu \mathbf{t} \oplus \{ A : \& (\{ A : \mathbf{t} \} \uplus \{ A' : T'' \}_{A' \in \Gamma \setminus \{A\}}) \}_{A \in \Gamma}$ which, after consuming a label A will add an input with label A , simulating the adding of A to the queue.
4. *The encoding of the finite control writes the wrong symbol.* In this last case, the finite control wishes to write a symbol to the queue with $\oplus (\{ B_1 : \llbracket B_2 \dots B_m \rrbracket_r^\tau \} \uplus \{ A' : T' \}_{A' \in \Gamma \setminus \{B_1\}})$. However, the simulation executes the wrong branch (with any $A' \neq B_1$) and continues as T' . In this case, T' continues removing and adding any value from the queue, indefinitely. Note that it is also possible that it removes the wrong value from the queue overlapping with case 2. We also observe that in this case the requirement

that the queue machine is single consuming is not necessary.

We can state the following result (and its corollary):

► **Theorem 24.** *Given a single consuming queue machine $M = (Q, \Sigma, \Gamma, \$, s, \delta)$ and an input string $x \in \Sigma^*$, we have $\llbracket s \rrbracket^0 \leq \llbracket x\$ \rrbracket$ if and only if M does not terminate on x .*

Below, subtyping without output covariance and input contravariance is defined as in Definition 4 except that $I = J_k$ in point 2. and $I = J$ in point 3.

► **Corollary 25.** *Subtyping without output covariance and input contravariance is undecidable.*

Notice that this also provides an alternative proof that asynchronous subtyping (Definition 4) is undecidable.

4.3 Bounded Asynchronous Subtyping

We conclude this section by defining bounded asynchronous subtyping and showing that it is undecidable.

► **Definition 26** (Bounded Asynchronous Subtyping, \leq_b). *The bounded asynchronous subtyping \leq_b is the union of all k -bounded asynchronous subtypings, for every $k \in \mathbb{N}$.*

The idea of bounded asynchronous subtyping can also be seen as bound to the size of the queue of a queue machine:

► **Definition 27** (Queue Machine Boundedness). *Let M be a queue machine and x a possible input. We say that M is bound on input x if there exists k such that, for every q and γ such that $(s, x\$) \rightarrow_M^* (q, \gamma)$, we have that $|\gamma| \leq k$.*

Boundedness of a queue machine is undecidable:

► **Lemma 28.** *Given a queue machine M and an input x , it is undecidable whether M terminates and is bound on x .*

We can then give a different encoding of queue machines into bounded subtyping:

► **Definition 29** (Queue Machine Encoding for Bounded Subtyping). *Let $M = (Q, \Sigma, \Gamma, \$, s, \delta)$ be a queue machine, and let $C_1, \dots, C_m \in \Gamma$, with $m \geq 0$, $q \in Q$ and $S \subseteq Q$. We define the $\llbracket C_1 \dots C_m \rrbracket$, resp. $\llbracket q \rrbracket^S$, in the same way as $\llbracket C_1 \dots C_m \rrbracket$, resp. $\llbracket q \rrbracket^S$, with the difference that the \uplus operator is interpreted in the following way: $\{l_i : T_i\}_{i \in I} \uplus \{l_j : T_j\}_{j \in J} = \{l_i : T_i\}_{i \in I}$.*

Finally, we can state the following impossibility result:

► **Theorem 30.** *Given a single consuming queue machine $M = (Q, \Sigma, \Gamma, \$, s, \delta)$ and an input string x , we have that $\llbracket s \rrbracket^0 \leq_b \llbracket x\$ \rrbracket$ if and only if M does not terminate and is bound on x .*

5 Related Work

The articles closest to ours are those by Bravetti et al. [1] and by Lange and Yoshida [8]. Concerning decidability results, both articles investigate fragments of session types for which subtyping becomes decidable, which however are much more limited, and far from having practical applications (they both require one of the compared types to have no branching at all), with respect to the fragments considered here. The former adopts a subtyping relation that allows orphan messages and consider two fragments: a fragment where type T , in $T \leq S$,

has no branching and S is single-out and another fragment where T is single-in and S has no branching. For both cases, they provide an algorithm by adding extra rules to a subtyping procedure similar to ours. For types that do not produce orphan messages, both fragments are just a special case of our algorithm for single-out and single-in session types. Lange and Yoshida consider, instead, a subtyping relation without orphan messages like ours, based on the original definition given by Chen et al. [2]. They give an algorithm for the case where one of the two types for which we wish to check subtyping has no branching. Note that our subtyping procedure is inspired by the work of Mostrous et al. [10] on asynchronous subtyping for multiparty session types.

The main result by Bravetti et. al [1] is the undecidability of asynchronous subtyping which is shown by using an encoding of queue machines. Lange and Yoshida [8] show undecidability of asynchronous subtyping by encoding Turing machines into a notion of subtyping for communicating automata. Then, such a result is transferred to session types. Unlike ours, both encodings take advantage of the use of input contravariance and output covariance. For example, by exploiting this feature, the queue machine encoding by Bravetti et al. is much simpler than ours. We note that our results on undecidability focus on binary session types. However, it is immediate to generalise this kind of undecidability results from binary session types to multiparty session types since binary session types are just multiparty session types with only two roles [1].

6 Conclusions

In this article, we have shed light on the boundaries between decidability and undecidability of asynchronous subtyping for session types. In particular, we have shown four main results: the decidability of k -bounded subtyping, the decidability of subtyping for single-out and single-in session types, the undecidability of subtyping without output covariance and input contravariance, and, finally, the undecidability of bounded subtyping.

From our results, it is clear that branching of inputs and outputs together with asynchrony (output anticipation) determine the boundaries between decidability and undecidability of asynchronous subtyping. Our goal for the future is to investigate whether other forms of restriction allow us to obtain a decidable relation, while retaining general branching for both inputs and outputs.

References

- 1 Mario Bravetti, Marco Carbone, and Gianluigi Zavattaro. Undecidability of asynchronous session subtyping. *CoRR*, abs/1611.05026, 2016. URL: <http://arxiv.org/abs/1611.05026>.
- 2 Tzu-Chun Chen, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. On the preciseness of subtyping in session types. In *16th International Symposium on Principles and Practice of Declarative Programming (PPDP'14)*, pages 135–146. ACM, 2014.
- 3 Simon J. Gay and Malcolm Hole. Subtyping for session types in the pi calculus. *Acta Inf.*, 42(2-3):191–225, 2005.
- 4 Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *7th European Symposium on Programming (ESOP'98)*, volume 1381 of *LNCS*, pages 122–138. Springer, 1998.

- 5 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2008)*, pages 273–284, 2008.
- 6 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *J. ACM*, 63(1):9, 2016.
- 7 Thomas Bracht Laumann Jespersen, Philip Munksgaard, and Ken Friis Larsen. Session types for rust. In *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming, WGP@ICFP 2015*, pages 13–22, 2015.
- 8 Julien Lange and Nobuko Yoshida. On the undecidability of asynchronous session subtyping. In *To appear in Proc. of FOSSACS 2017*, 2017.
- 9 Sam Lindley and J. Garrett Morris. Embedding session types in haskell. In *Proceedings of the 9th International Symposium on Haskell (Haskell 2016)*, pages 133–145, 2016.
- 10 Dimitris Mostrous, Nobuko Yoshida, and Kohei Honda. Global principal typing in partially commutative asynchronous sessions. In *18th European Symposium on Programming (ESOP’09)*, volume 5502 of *LNCS*, pages 316–332. Springer, 2009.
- 11 Nicholas Ng and Nobuko Yoshida. Static deadlock detection for concurrent go by global session graph synthesis. In *Proceedings of the 25th International Conference on Compiler Construction (CC 2016)*, pages 174–184, 2016.

A

 Proofs of Section 2

A.1 Proof of Theorem 6

► **Lemma 31.** *Given two session types T and S , we have that $T \leq S$ implies $T \leq_o S$.*

Proof. Given an asynchronous subtyping relation \mathcal{R} we show that \mathcal{R} is also an orphan-message-free subtyping relation. To this aim we need to prove that if $(T, S) \in \mathcal{R}$ and $T = \oplus\{l_i : T_i\}_{i \in I}$ then besides all the items in 2. of Definition 4 we also have:

- if $\mathcal{A} \neq []^1$ then $\forall i \in I. \& \in T_i$.

From $\mathcal{A} \neq []^1$ it follows that S (that corresponds with $\mathcal{A}[S_{ki}]^{k \in \{1, \dots, m\}}$) starts with an input. As \mathcal{R} is an asynchronous subtyping relation, it is dual closed, hence $(\bar{S}, \bar{T}) \in \mathcal{R}$. We observe that \bar{S} starts with an output and $\bar{T} = \&\{l_i : \bar{T}_i\}_{i \in I}$. For item 2. of Definition 4, we have that $\bar{T} = \mathcal{A}'[\oplus\{l_j : V_{kj}\}_{j \in J_k}]^{k \in \{1, \dots, m\}}$, for some input context \mathcal{A}' . This means that all \bar{T}_i contain at least an internal choice, which implies that all T_i contain at least one external choice. ◀

► **Lemma 32.** *Given two session types T and S , we have that $T \leq_o S$ implies $T \leq S$.*

Proof. We show that, given $T \leq_o S$, it is possible to define an asynchronous subtyping relation \mathcal{R} s.t. $(T, S) \in \mathcal{R}$. Consider

$$\mathcal{R} = \{(T, S), (\bar{S}, \bar{T}) \mid T \leq_o S\}$$

The relation \mathcal{R} is dual closed by definition. It remains to show that it satisfies the four items in Definition 4. Let $(T, S) \in \mathcal{R}$. There are two cases: $T \leq_o S$ or $\bar{S} \leq_o \bar{T}$. In the first case all the item holds by definition of orphan-message-free subtyping relation. We consider now the second case, i.e. $\bar{S} \leq_o \bar{T}$, and proceeds with a case analysis.

1. $T = \text{end}$.

We have $\bar{T} = \text{end}$. Having $\bar{S} \leq_o \text{end}$, by definition of \leq_o , in particular by n applications of item 4. (with $n \geq 0$) and one application of item 1., it follows that $\bar{S} = \mu t_1 \dots \mu t_n. \text{end}$. Hence $S = \mu t_1 \dots \mu t_n. \text{end}$, then we can conclude what requested, i.e., $\exists n \geq 0$ such that $\text{unfold}^n(S) = \text{end}$.

2. $T = \oplus\{l_i : T_i\}_{i \in I}$.

We have $\bar{T} = \&\{l_i : \bar{T}_i\}_{i \in I}$. Having $\bar{S} \leq_o \&\{l_i : \bar{T}_i\}_{i \in I}$, by definition of \leq_o , we have two possible cases.

a. By n applications of item 4. (with $n \geq 0$) and one application of item 3., it follows that $\bar{S} = \mu t_1 \dots \mu t_n. \&\{l_j : \bar{S}_j\}_{j \in J}$, with $J \subseteq I$ and $\text{unfold}^n(\bar{S}) = \&\{l_j : \bar{S}'_j\}_{j \in J}$ with $\bar{S}'_j \leq_o \bar{T}_j$ for every $j \in J$. Hence $S = \mu t_1 \dots \mu t_n. \oplus\{l_j : S_j\}_{j \in J}$, then we can conclude what requested, i.e., $\text{unfold}^n(S) = [\oplus\{l_j : S'_j\}_{j \in J}]^1$, $J \subseteq I$ and $\forall j \in J. (T_j, S'_j) \in \mathcal{R}$. Notice that we have used the fact that $\text{unfold}^n(\bar{S}) = \text{unfold}^n(S)$ and we have considered an input context $\mathcal{A} = []^1$.

b. By n applications of item 4. (with $n \geq 0$) and one application of item 2., it follows that $\bar{T} = \&\{l_i : \bar{T}_i\}_{i \in I} = \mathcal{A}[\oplus\{l_p : \bar{T}_{kp}\}_{p \in J_k}]^{k \in \{1, \dots, m\}}$ (hence with $\neq []^1$), and $\bar{S} = \mu t_1 \dots \mu t_n. \oplus\{l_j : \bar{S}_j\}_{j \in J}$, with $\forall k \in \{1, \dots, m\}. J \subseteq J_k$ and $\text{unfold}^n(\bar{S}) = \&\{l_j : \bar{S}'_j\}_{j \in J}$ with $\forall j \in J. \bar{S}'_j \leq_o \mathcal{A}[\bar{T}_{kj}]^{k \in \{1, \dots, m\}}$. Hence $S = \mu t_1 \dots \mu t_n. \&\{l_j : S_j\}_{j \in J}$. We now observe that there exists an input context \mathcal{A}' and n', m' such that $\text{unfold}^{n'}(S) = \mathcal{A}'[\oplus\{l_m : S_{km}\}_{m \in L_k}]^{k \in \{1, \dots, m'\}}$ with $\forall k \in \{1, \dots, m'\}. I \in L_k$. This follows from the fact that $\bar{S} \leq_o \&\{l_i : \bar{T}_i\}_{i \in I}$: by repeated application of the rule 2. of Definition 4, and its additional constraint imposed by Definition 5, we have the guarantee that along all branches of \bar{S} (and its unfoldings) it is guaranteed to

reach an external choice, and by application of rule 3. (in particular the contravariance on external choices), the labels of such choices include the set of labels of the initial external choice of $\&\{l_i : \bar{T}_i\}_{i \in I}$. We conclude by showing that what is requested, i.e., $\forall i \in I. (T_i, \mathcal{A}'[S_{ki}]^{k \in \{1, \dots, m'\}}) \in \mathcal{R}$, actually holds. This follows from the fact that $\mathcal{A}'[S_{ki}]^{k \in \{1, \dots, m'\}} \leq_o \bar{T}_i$, which is a consequence of $\bar{S} \leq_o \bar{T}$. In fact, this implies that also $\text{unfold}^n(\bar{S}) \leq_o \bar{T}$ because an orphan-message-free subtyping relation is still such even if we add pairs $(\text{unfold}^n(V), Z)$ assuming (V, Z) already in the relation. Having $\text{unfold}^n(\bar{S}) = \text{unfold}^n(\bar{S}) = \mathcal{A}'[\&\{l_m : \bar{S}_{km}\}_{m \in L_k}]^{k \in \{1, \dots, m'\}}$ and $\bar{T} = \&\{l_i : \bar{T}_i\}_{i \in I}$, it is easy to see that given an orphan-message-free subtyping relation \mathcal{R}' such that $(\mathcal{A}'[\&\{l_m : \bar{S}_{km}\}_{m \in L_k}]^{k \in \{1, \dots, m'\}}, \&\{l_i : \bar{T}_i\}_{i \in I}) \in \mathcal{R}'$, the relation obtained by enriching \mathcal{R}' with the pairs $(\mathcal{A}''[\bar{S}_{ki}]^{k \in K \subseteq \{1, \dots, m'\}}, \bar{T}'_i)$ assuming $(\mathcal{A}''[\&\{l_m : \bar{S}_{km}\}_{m \in L_k}]^{k \in K \subseteq \{1, \dots, m'\}}, \&\{l_i : \bar{T}'_i\}_{i \in I}) \in \mathcal{R}'$, is still an orphan-message-free subtyping relation. Above we adopt an abuse of notation for input contexts: $\bar{B}[W_k]^{k \in K \subseteq \{1, \dots, t\}}$ does not have holes numbered consistently from 1 to t , but some numbers in $\{1, \dots, t\}$ could be missing.

3. $T = \&\{l_i : T_i\}_{i \in I}$.

We have $\bar{T} = \oplus\{l_i : \bar{T}_i\}_{i \in I}$. Having $\bar{S} \leq_o \oplus\{l_i : \bar{T}_i\}_{i \in I}$, by definition of \leq_o , in particular by n applications of item 4. (with $n \geq 0$) and one application of item 2., it follows that $\bar{S} = \mu t_1 \dots \mu t_n. \oplus\{l_j : \bar{S}_j\}_{j \in J}$, with $J \subseteq I$, and $\text{unfold}^n(\bar{S}) = \oplus\{l_j : \bar{S}'_j\}_{j \in J}$ with $\bar{S}'_j \leq_o \bar{T}_j$ for every $j \in J$. Hence $S = \mu t_1 \dots \mu t_n. \&\{l_j : S_j\}_{j \in J}$, then we can conclude what requested, i.e., $\text{unfold}^n(S) = \&\{l_j : S'_j\}_{j \in J}$, $J \subseteq I$ and $\forall j \in J. (T_j, S'_j) \in \mathcal{R}$. Notice that we have used the fact that $\text{unfold}^n(\bar{S}) = \text{unfold}^n(\bar{S})$.

4. $T = \mu t. T'$.

We first observe that $V \leq_o \mu t. Z$ implies $V \leq_o Z\{\mu t. Z/t\}$. This directly follows from the fact that if $(V, \mu t. Z)$ belongs to an orphan-message-free subtyping relation, then the same relation enriched with the pair $(V, Z\{\mu t. Z/t\})$ is still an orphan-message-free subtyping relation. We now proceed by considering $\bar{T} = \mu t. \bar{T}'$. As $\bar{S} \leq_o \bar{T}$, we have $\bar{S} \leq_o \mu t. \bar{T}'$. By the above observation we have $\bar{S} \leq_o \bar{T}'\{\mu t. \bar{T}'/t\}$ that implies what requested, i.e., $(T'\{\mu t. T'/t\}, S) \in \mathcal{R}$. \blacktriangleleft

► **Theorem 6.** *Given two session types S and T , we have $T \leq S$ if and only if $T \leq_o S$.*

Proof. Direct consequence of Lemmas 31 and 32. \blacktriangleleft

B Proofs of Section 3

B.1 Proof of Proposition 10

► **Proposition 10.** *Given the types T and S we have that there exist Σ', T', S' , such that $\emptyset \vdash T \leq_a S \rightarrow^* \Sigma' \vdash T' \leq_a S' \rightarrow_{\text{err}}$, if and only if $T \not\leq S$*

Proof. We prove the two implications separately.

We start with the *if* part and proceed by contraposition. We assume that it is not true that $\exists \Sigma', T', S'. \emptyset \vdash T \leq_a S \rightarrow^* \Sigma' \vdash T' \leq_a S' \rightarrow_{\text{err}}$ and show that $T \leq S$. We first observe that even if we remove rule **Asmp** from the procedure, it is still impossible to reach a judgement $\Sigma' \vdash T' \leq_a S'$ on which no rule can be applied. Let $\rightarrow_{\text{noAsmp}}$ be our decision procedure under the assumption that **Asmp** cannot be used. By contraposition, assume $\emptyset \vdash T \leq_a S \rightarrow_{\text{noAsmp}}^* \Sigma' \vdash T' \leq_a S' \rightarrow_{\text{err}}$. We have that there exists an intermediary judgement $\Sigma'' \vdash T'' \leq_a S''$ such that $\emptyset \vdash T \leq_a S \rightarrow^* \Sigma'' \vdash T'' \leq_a S''$ (notice the use of the standard procedure), $(T'', S'') \in \Sigma''$ and $\Sigma'' \vdash T'' \leq_a S'' \rightarrow_{\text{noAsmp}}^* \Sigma' \vdash T' \leq_a S'$. Within

the sequence of rule applications $\Sigma'' \vdash T'' \leq_a S'' \rightarrow_{noAsmp}^* \Sigma' \vdash T' \leq_a S'$ we consider the judgement $\Sigma''' \vdash T''' \leq_a S'''$ which is the last one such that $(T''', S''') \in \Sigma''$ (such judgement exists as the first one $\Sigma'' \vdash T'' \leq_a S''$ already has this property). It is not restrictive to assume that in the sequence $\Sigma''' \vdash T''' \leq_a S''' \rightarrow_{noAsmp}^* \Sigma' \vdash T' \leq_a S'$ there is no two judgements $\Sigma_1 \vdash T_1 \leq_a S_1$ and $\Sigma_2 \vdash T_2 \leq_a S_2$ with $T_1 = T_2$ and $S_1 = S_2$ (otherwise we can shorten the sequence $\Sigma''' \vdash T''' \leq_a S''' \rightarrow_{noAsmp}^* \Sigma' \vdash T' \leq_a S'$ obtaining a new one having the same properties). Consider now, in the standard application of the procedure $\emptyset \vdash T \leq_a S \rightarrow^* \Sigma'' \vdash T'' \leq_a S''$, the intermediary judgement $\Sigma_i \vdash T''' \leq_a S'''$ that added (T''', S''') to the environment; we have that from this judgement there exists a standard application of the procedure $\emptyset \vdash T \leq_a S \rightarrow^* \Sigma_i \vdash T''' \leq_a S''' \rightarrow^* \Sigma'_i \vdash T'''' \leq_a S'''' \rightarrow_{err}$ simply by considering from $\Sigma_i \vdash T''' \leq_a S'''$ the same rules used in the sequence $\Sigma''' \vdash T''' \leq_a S''' \rightarrow_{noAsmp}^* \Sigma' \vdash T' \leq_a S'$.

Consider now the relation $\mathcal{R} = \{(T', S') \mid \exists \Sigma'. \Sigma' \vdash T' \leq_a S' \in \mathcal{S}\}$ where \mathcal{S} is the minimal set of judgements satisfying the following:

- $\emptyset \vdash T \leq_a S \in \mathcal{S}$;
- if $\Sigma' \vdash T' \leq_a S' \in \mathcal{S}$ and $\Sigma' \vdash T' \leq_a S' \in \mathcal{S} \rightarrow \Sigma'' \vdash T'' \leq_a S''$, without applying rule **Asmp** or **RecR₂**, then $\Sigma'' \vdash T'' \leq_a S'' \in \mathcal{S}$;
- if $\Sigma' \vdash T' \leq_a S' \in \mathcal{S}$ and $\Sigma' \vdash T' \leq_a S' \in \mathcal{S} \rightarrow \Sigma'' \vdash T'' \leq_a S''$ by applying **RecR₂**, then $\Sigma'' \vdash T'' \leq_a \text{unfold}^{\text{outDepth}(S')}(S') \in \mathcal{S}$.

We observe that to each judgement $\Sigma' \vdash T' \leq_a S' \in \mathcal{S}$ it is always possible to apply at least one rule. In fact, if this is not possible, we would have also $\emptyset \vdash T \leq_a S \rightarrow_{noAsmp}^* \Sigma'' \vdash T'' \leq_a S'' \rightarrow_{err}$ for a judgement $\Sigma'' \vdash T'' \leq_a S''$ with $T'' = T'$ and S'' less unfolded than S' . In fact, the unique difference between the judgements in \mathcal{S} and those reachable without adopting **Asmp** is that those in \mathcal{S} are more unfolded (see the difference between $\text{outUnf}(S)$ used in rule **RecR₂** and $\text{unfold}^{\text{outDepth}(S')}(S')$ used in the definition of \mathcal{S}).

We finally show that \mathcal{R} is an orphan-message-free subtyping relation, hence $T \leq_o S$ that, by Theorem 6, implies also $T \leq S$. Let $(T', S') \in \mathcal{R}$. Then $\Sigma' \vdash T' \leq_a S' \in \mathcal{S}$ and it is possible to apply at least one rule to $\Sigma' \vdash T' \leq_a S'$. We proceed by cases on T' .

- If $T' = \mathbf{end}$ then item 1. of Definition 4 for pair (T', S') is shown by induction on $k = \text{nrec}(S')$, i.e. the number of unguarded (not prefixed by some input or output) occurrences of recursions $\mu t. S''$ in S' for any S'', t .
 - Base case $k = 0$. The only rule applicable to $\Sigma' \vdash T' \leq_a S'$ is **End**, that immediately yields the desired pair of \mathcal{R} .
 - Induction case $k > 0$. The only rules applicable to $\Sigma' \vdash T' \leq_a S'$ are **Asmp** and **RecR₁**. In the case of **Asmp** we have that $(T', S') \in \Sigma'$, hence there exists Σ'' with $(T', S') \notin \Sigma''$ such that $\Sigma'' \vdash T' \leq_a S' \in \mathcal{S}$. **RecR₁** can be applied to $\Sigma'' \vdash T' \leq_a S'$. So for some $\Sigma''' (= \Sigma'$ or $= \Sigma'')$ we have that the procedure applies rule **RecR₁** to $\Sigma''' \vdash T' \leq_a S'$. Hence $\Sigma''' \vdash T' \leq_a S' \rightarrow \Sigma'''' \vdash T' \leq_a \text{unfold}^1(S')$. Since $\text{nrec}(\text{unfold}^1(S')) = k - 1$, by induction hypothesis item 1. of Definition 4 holds for pair $(T', \text{unfold}^1(S'))$, hence it holds for pair (T', S') .
- If $T' = \oplus \{l_i : T_i\}_{i \in I}$ then item 2. of Definition 4 for pair (T', S') is shown as follows.
 - If $\text{outDepth}(S') = 0$ then the only rule applicable to $\Sigma' \vdash T' \leq_a S'$ is **Out**, that immediately yields the desired pairs of \mathcal{R} .
 - If $\text{outDepth}(S') \geq 1$ then the only rules applicable to $\Sigma' \vdash T' \leq_a S'$ are **Asmp** and **RecR₂**. In the case of **Asmp** we have that $(T', S') \in \Sigma'$, hence there exists Σ'' with $(T', S') \notin \Sigma''$ such that $\Sigma'' \vdash T' \leq_a S' \in \mathcal{S}$. **RecR₂** can be applied to $\Sigma'' \vdash T' \leq_a S'$. So for some $\Sigma''' (= \Sigma'$ or $= \Sigma'')$ we have that the procedure applies rule **RecR₂** to $\Sigma''' \vdash T' \leq_a S'$. Hence $(T', \text{unfold}^{\text{outDepth}(S')}(S')) \in \mathcal{R}$. Since

- $\text{outDepth}(\text{unfold}^{\text{outDepth}(S')}(S')) = 0$, we end up in the previous case. Therefore item 2. of Definition 4 holds for pair $(T', \text{unfold}^{\text{outDepth}(S')}(S'))$, hence it holds for pair (T', S') .
- If $T' = \&\{l_i : T_i\}_{i \in I}$ then item 3. of Definition 4 for pair (T', S') is shown by induction on $k = \text{nrec}(S')$.
 - Base case $k = 0$. The only rule applicable to $\Sigma' \vdash T' \leq_a S'$ is **In**, that immediately yields the desired pairs of \mathcal{R} .
 - Induction case $k > 0$. The only rules applicable to $\Sigma' \vdash T' \leq_a S'$ are **Asmp** and **RecR₁**. In the case of **Asmp** we have that $(T', S') \in \Sigma'$, hence there exists Σ'' with $(T', S') \notin \Sigma''$ such that $\Sigma'' \vdash T' \leq_a S' \in \mathcal{S}$. **RecR₁** can be applied to $\Sigma'' \vdash T' \leq_a S'$. So for some Σ''' ($= \Sigma'$ or $= \Sigma''$) we have that the procedure applies rule **RecR₁** to $\Sigma''' \vdash T' \leq_a S'$. Hence $\Sigma''' \vdash T' \leq_a S' \rightarrow \Sigma'''' \vdash T' \leq_a \text{unfold}^1(S')$. Since $\text{nrec}(\text{unfold}^1(S')) = k - 1$, by induction hypothesis item 3. of Definition 4 holds for pair $(T', \text{unfold}^1(S'))$, hence it holds for pair (T', S') .
 - If $T' = \mu t.T''$ then item 4. of Definition 4 for pair (T', S') holds because the only rule applicable to $\Sigma' \vdash T' \leq_a S'$ is **Recl** that immediately yields the desired pair of \mathcal{R} .

We now prove the *only if* part and proceed by contraposition. We assume that $T \leq S$ and show that there exist no Σ', T', S' , such that $\emptyset \vdash T \leq_a S \rightarrow^* \Sigma' \vdash T' \leq_a S' \rightarrow_{\text{err}}$. If $T \leq S$ then also $T \leq_o S$ (by Theorem 6). So we can assume the existence of a relation \mathcal{R} that is an orphan-message-free subtyping relation such that $(T, S) \in \mathcal{R}$.

We say that $\Sigma \vdash T \leq_a S \rightarrow_w \Sigma' \vdash T' \leq_a S'$ if $\Sigma \vdash T \leq_a S \rightarrow^* \Sigma' \vdash T' \leq_a S'$ and: the last rule applied is one of **Out**, **In** or **Recl** rules; while all previous ones are **RecR₁** or **RecR₂** rules. As another notation we use input-output-end contexts \mathcal{B} defined as the input contexts in Definition 3 with the difference that also the output construct and **end** are part of the grammar in the definition.

We start by showing that $\exists \Sigma. \emptyset \vdash T \leq_a S \rightarrow_w^* \Sigma \vdash T' \leq_a S'$ implies $S' = \mathcal{B}[S_k]^{k \in \{1 \dots m\}}$, $S_k = \mu t_k.S'_k$, for some t_k and S'_k , and $\exists n_1, \dots, n_m. (T', \mathcal{B}[\text{unfold}^{n_k}(S_k)]^{k \in \{1 \dots m\}}) \in \mathcal{R}$. The proof is by induction on the length of such computation \rightarrow_w^* . The base case is for a 0 length computation: it yields $(T, S) \in \mathcal{R}$ which holds. For the inductive case we assume it to hold for all computations of a length k and we show it to hold for all computations of length $k + 1$, by considering all judgements $\Sigma' \vdash T'' \leq_a S''$ such that $\Sigma \vdash T' \leq_a S' \rightarrow_w \Sigma' \vdash T'' \leq_a S''$. This is shown by first considering the case in which rule **Asmp** applies to $\Sigma \vdash T' \leq_a S'$: in this case there is no such a judgement and there is nothing to prove. Then we consider the case in which $T' = \text{end}$ and $\Sigma \vdash \text{end} \leq_a S' \rightarrow^* \Sigma''' \vdash \text{end} \leq_a \text{end}$ (by applying **RecR₁** rules) and rule **End** applies to $\Sigma''' \vdash \text{end} \leq_a \text{end}$. Also in this case there is no such a judgement $\Sigma' \vdash T'' \leq_a S''$ and there is nothing to prove. Finally, we proceed by an immediate verification that judgements $\Sigma' \vdash T'' \leq_a S''$ produced in remaining cases are required to be in \mathcal{R} by items 2., 3. and 4. of Definition 4: $T' = \oplus\{l_i : T_i\}_{i \in I}$ (\rightarrow_w is a possibly empty sequence of **RecR₂** applications followed by **Out** application), $T' = \&\{l_i : T_i\}_{i \in I}$ (\rightarrow_w is a possibly empty sequence of **RecR₁** applications followed by **In** application) or $T' = \mu t.T''$ (\rightarrow_w is simply **Recl** application).

We finally observe that, given a judgement $\Sigma \vdash T' \leq_a S'$ such that $S' = \mathcal{B}[S_k]^{k \in \{1 \dots m\}}$, $S_k = \mu t_k.S'_k$, for some t_k and S'_k , and $\exists n_1, \dots, n_m. (T', \mathcal{B}[\text{unfold}^{n_k}(S_k)]^{k \in \{1 \dots m\}}) \in \mathcal{R}$ we have:

- either rule **Asmp** applies to $\Sigma \vdash T' \leq_a S'$, or
- $T' = \text{end}$ and, by item 1. of Definition 4, there exists Σ' such that $\Sigma \vdash \text{end} \leq_a S' \rightarrow^* \Sigma' \vdash \text{end} \leq_a \text{end}$ (by applying **RecR₁** rules) and rule **End** is the unique rule applicable to $\Sigma' \vdash \text{end} \leq_a \text{end}$, with **RecR₁** being the unique rule applicable to intermediate judgements, or

- by items 2., 3. and 4. of Definition 4, there exist Σ', T'', S'' such that $\Sigma \vdash T' \leq_a S' \rightarrow_w^*$ $\Sigma' \vdash T'' \leq_a S''$, with each intermediate judgement having a unique applicable rule. In particular this holds for $T' = \oplus\{l_i : T_i\}_{i \in I}$ (\rightarrow_w is a possibly empty sequence of RecR_2 applications followed by Out application), $T' = \&\{l_i : T_i\}_{i \in I}$ (\rightarrow_w is a possibly empty sequence of RecR_1 applications followed by In application) or $T' = \mu t.T'$ (\rightarrow_w is simply RecL application). \blacktriangleleft

B.2 Proof of Theorem 12

► **Theorem 12.** *The algorithm for \leq_a^k always terminates and, given the types T and S , there exist Σ', T', S' such that $\emptyset \vdash T \leq_a^k S \rightarrow^* \Sigma' \vdash T' \leq_a S' \rightarrow_{\text{err}}$ if and only if $T \not\leq_k S$.*

Proof. We first observe that the decision algorithm for k -bounded asynchronous subtyping terminates. By contraposition, if the algorithm does not terminate, there exists an infinite sequence $\Sigma \vdash T \leq_a S \rightarrow \Sigma_1 \vdash T_1 \leq_a S_1 \rightarrow^* \Sigma_i \vdash T_i \leq_a S_i \rightarrow^*$. Along this infinite sequence infinitely many distinct pairs (T, S) will be added to Σ . As only finitely many distinct terms can be reached as first element of the pairs, there will be infinitely many distinct terms as second element. Such terms will have unbounded depth, but this is not possible due to the constraint added to rule Out that impose the use of k -bounded input contexts.

We now prove that, given the types T and S , there exist Σ', T', S' such that $\emptyset \vdash T \leq_a^k S \rightarrow^* \Sigma' \vdash T' \leq_a S' \rightarrow_{\text{err}}$ if and only if $T \not\leq_k S$.

We start with the *if* part and proceed by contraposition. We assume that it is not true that $\exists \Sigma', T', S'. \emptyset \vdash T \leq_a^k S \rightarrow^* \Sigma' \vdash T' \leq_a S' \rightarrow_{\text{err}}$ and we build a relation \mathcal{R} that we show to be a k -bounded Asynchronous Subtyping relation. The relation \mathcal{R} is built from the judgments $\Sigma'' \vdash T'' \leq_a^k S''$ exactly as we did for the \leq_a subtyping procedure in the first part (the *if* part) of the proof of Proposition 10. In such a proof we show \mathcal{R} to be an orphan-message-free subtyping relation, hence we just have to show it to be k -bounded. It is immediate to observe that, since when applying rule Out to a judgment $\Sigma'' \vdash T'' \leq_a^k S''$ we require the input context \mathcal{A} to be k -bounded, we may include in \mathcal{R} only pairs (T'', S'') that satisfy the same constraint in item 2 of k -bounded Asynchronous Subtyping relation definition (Definition 11), because otherwise we would have $\Sigma'' \vdash T'' \leq_a^k S'' \rightarrow^* \Sigma''' \vdash T''' \leq_a^k S''' \rightarrow_{\text{err}}$ by possibly applying $\text{RecR}_1/\text{RecR}_2$ rules. Hence, as justified in Proposition 10 this would lead to violating the assumption that the algorithm does not reach an error. The justification provided there still holds because judgments $\Sigma'' \vdash T'' \leq_a^k S''_1$ and $\Sigma'' \vdash T'' \leq_a^k S''_2$, with S''_1 and S''_2 that just differ for the level of internal unfoldings, behave equivalently with respect to errors due to k -boundedness violations. This because the k -boundedness of context \mathcal{A} is established by the Out rule after unfolding in S''_1/S''_2 all recursions occurring before the first output of every possible branch by means of the $\text{RecR}_1/\text{RecR}_2$ rules.

We now prove the *only if* part and proceed by contraposition. We assume that $T \leq_k S$ and show that there exist no Σ', T', S' , such that $\emptyset \vdash T \leq_a^k S \rightarrow^* \Sigma' \vdash T' \leq_a S' \rightarrow_{\text{err}}$. If $T \leq_k S$ then also $T \leq_o S$. So we can assume the existence of a relation \mathcal{R} that is an orphan-message-free subtyping relation such that $(T, S) \in \mathcal{R}$. We then use exactly the same proof as that of the second part (the *only if* part) of the proof of Proposition 10 to establish a correspondence between judgements $\Sigma'' \vdash T'' \leq_a^k S''$, such that $\emptyset \vdash T \leq_a^k S \rightarrow_w^* \Sigma'' \vdash T'' \leq_a^k S''$, and pairs in \mathcal{R} (see the construction of the corresponding pair in the proof of Proposition 10). Since \mathcal{R} includes only pairs that satisfy the constraint in item 2 of k -bounded Asynchronous Subtyping relation definition (Definition 11) requiring context \mathcal{A} to be k -bounded; and since any judgment $\Sigma'' \vdash T'' \leq_a^k S''$ such that $\emptyset \vdash T \leq_a^k S \rightarrow_w^* \Sigma'' \vdash T'' \leq_a^k S''$ implies there is in \mathcal{R} a corresponding pair (T'', S''_1) , with S''_1 differing from S'' just for the level

of internal unfoldings, we have that reachable judgments $\Sigma'' \vdash T'' \leq_a^k S''$ cannot be such that: $\Sigma'' \vdash T'' \leq_a^k S'' \rightarrow^* \Sigma''' \vdash T''' \leq_a^k S'''$, by possibly applying $\text{RecR}_1/\text{RecR}_2$ rules, and $\Sigma''' \vdash T''' \leq_a^k S''' \rightarrow_{\text{err}}$ due to not satisfying the requirement about the input context \mathcal{A} to be k -bounded in the rule **Out**. This because the difference in unfolding levels between S'' and S'_1 (inside judgment $\Sigma'' \vdash T'' \leq_a^k S''$ and the corresponding pair (T'', S'_1) in \mathcal{R}) is not significant: the k -boundedness of context \mathcal{A} is established both in the rule **Out** and in item 2 of \leq_k definition after unfolding all recursions occurring before the first output of every possible branch.

This observation makes it possible to carry out the proof as in Proposition 10, hence to show that there exist no Σ', T', S' , such that $\emptyset \vdash T \leq_a^k S \rightarrow^* \Sigma' \vdash T' \leq_a^k S' \rightarrow_{\text{err}}$. \blacktriangleleft

B.3 Proof of Proposition 16

► **Lemma 33.** *Given a single-out session type T , $\text{reach}(T)$ is finite.*

Proof. We now define a finite set of session types $\text{fin}(T)$, and then we prove that it satisfies all the constraints 1., ..., 4. in Definition 15. Hence $\text{reach}(T) \subseteq \text{fin}(T)$ by definition, from which finiteness of $\text{reach}(T)$ follows.

It is not restrictive to assume that all the recursion variables of T are distinct: let $\mathbf{x}_1, \dots, \mathbf{x}_n$ be such variables. We consider the rewriting variables X_1, \dots, X_n . Let T_i be such that $\mu\mathbf{x}_i.T_i$ occurs in T ; let T' be T with X_i that replaces $\mu\mathbf{x}_i.T_i$; and similarly let T'_i be T_i with X_j that replaces each occurrence of $\mu\mathbf{x}_j.T_j$ and \mathbf{x}_j . We now consider the rewriting rules $X_i \rightarrow_i^1 T'_i$ and $X_i \rightarrow_i^2 \mathbf{x}_i$. Given one of the above term S containing rewriting variables, we denote with $\text{close}(S)$ the session type obtained by repeated application of the rewriting rules in the following way: if X_i occurs inside a subterm $\mu\mathbf{x}_i.S'$ apply \rightarrow_i^2 , otherwise apply \rightarrow_i^1 . We now define another closure function on sets of terms \mathcal{S} : $\text{subterms}(\mathcal{S}) = \{S' \mid S' \text{ is a subterm of } S \in \mathcal{S}\}$. Consider finally $\text{fin}(T) = \{\text{close}(S) \mid S \in \text{subterms}(\{T', T'_1, \dots, T'_n\})\}$. We have that $\text{fin}(T)$ is finite and it satisfies all the constraints 1., ..., 4. in Definition 15. \blacktriangleleft

We now report some definitions and results used in the proof of Proposition 16.

► **Definition 34.** Let T be a single-out session type. A relation \mathcal{R} over $\text{reach}(T)$ is an antEq_T relation if $(T', T'') \in \mathcal{R}$ implies: there exist $l, \mathcal{A}', \mathcal{A}''$ such that $\text{outUnf}(T') = \mathcal{A}'[\oplus\{l : T'_i\}]^{i \in \{1, \dots, n\}}$ and $\text{outUnf}(T'') = \mathcal{A}''[\oplus\{l : T''_j\}]^{j \in \{1, \dots, m\}}$, with $(T'_i, T''_j) \in \mathcal{R}$ for all $i \in \{1, \dots, n\}$ and $j \in \{1, \dots, m\}$. We say that $T' \text{ antEq}_T T''$ if there is an antEq_T relation \mathcal{R} such that $(T', T'') \in \mathcal{R}$.

Notice that antEq_T itself is an antEq_T relation because, obviously, the union of two antEq_T relations is an antEq_T relation and $\text{reach}(T)$ is finite. Moreover notice that, given a term $T' \in \text{reach}(T)$, all terms T'_i (with $i \in \{1, \dots, n\}$) for which $\text{outUnf}(T') = \mathcal{A}'[\oplus\{l : T'_i\}]^{i \in \{1, \dots, n\}}$ are always such that $T'_i \in \text{reach}(T)$ as well (because $\text{outUnf}(T')$ never unfolds recursions occurring inside terms T'_i). Finally, notice that antEq_T is decidable in that it is a relation over $\text{reach}(T)$, which is a finite set.

► **Definition 35.** antSet_T is the field of antEq_T , that is the set of session types $T' \in \text{reach}(T)$ such that there exists T'' with $(T', T'') \in \text{antEq}_T$ or $(T'', T') \in \text{antEq}_T$.

► **Lemma 36.** antEq_T is an equivalence relation on antSet_T .

Proof. The reflexive, symmetric and transitive closure of an antEq_T relation is an antEq_T relation, hence this holds true for antEq_T as well. \blacktriangleleft

► **Lemma 37.** *Let $T' \in \text{reach}(T)$. We have that $\text{antOutInf}(T')$ if and only if $T' \in \text{antSet}_T$.*

Proof. We prove the two implications separately, starting from the *if* part, e.g. by assuming $T' \in \text{antSet}_T$. By Lemma 36 we have $T' \text{ antEq}_T T'$. We now prove by induction on m that for every m there exists $l_{i_1} \cdots l_{i_m}$ such that $\text{antOut}(T', l_{i_1} \cdots l_{i_m})$ is defined. If $m = 1$ it is sufficient to consider $l_{i_1} = l$ where $\text{outUnf}(T') = \mathcal{A}'[\oplus\{l : T'_i\}]^{i \in \{1, \dots, n\}}$ (with \mathcal{A}' and T'_i that exist by Definition 34). Consider now that $T'' = \text{antOut}(T', l_{i_1} \cdots l_{i_{m-1}})$ is defined. By Definition 14, we have $T'' = \mathcal{A}[T_k]^k$ with $\text{outUnf}(\text{antOut}(T, l_{i_1} \cdots l_{i_{m-2}})) = \mathcal{A}[\oplus\{l_{i_{m-1}} : T_k\}]^k$. As $T' \text{ antEq}_T T'$, we can apply $m - 1$ times Definition 34 to conclude that $T_i \text{ antEq}_T T_j$, for every $i, j \in 1 \dots k$. This guarantees the existence of the input contexts \mathcal{A}^k , session types T_r^k , and label l such that $\text{outUnf}(T_k) = \mathcal{A}^k[\oplus\{l : T_r^k\}]^r$. This implies that it is possible to define $\text{antOut}(T'', l)$ hence also $\text{antOut}(T', l_{i_1} \cdots l_{i_m})$ by taking $l_{i_m} = l$.

We now move to the *only if* part assuming that there exists an infinite label sequence $l_{i_1} \cdots l_{i_n} \cdots$ such that, for every n , $\text{antOut}(T', l_{i_1} \cdots l_{i_n})$ is defined. Let \mathcal{R} be the minimal relation such that $(T', T') \in \mathcal{R}$ and: $\text{outUnf}(\text{antOut}(T', l_{i_1} \cdots l_{i_{n-1}})) = \mathcal{A}[\oplus\{l_{i_n} : T_k\}]^{k \in \{1 \dots m_n\}}$, for any $n \geq 1$, implies $\forall i, j \in \{1 \dots m_n\}. (T_i, T_j) \in \mathcal{R}$. We now show that \mathcal{R} above is an antEq_T relation. Considered any (T'', T''') in \mathcal{R} , we have that there exists h , with $h \geq 1$, such that, for some $\mathcal{A}', \mathcal{A}''$, we have: $\text{outUnf}(T'') = \mathcal{A}'[\oplus\{l_{i_h} : T'_i\}]^{i \in \{1, \dots, m'\}}$ and $\text{outUnf}(T''') = \mathcal{A}''[\oplus\{l_{i_h} : T''_j\}]^{j \in \{1, \dots, m''\}}$, with $(T'_i, T''_j) \in \mathcal{R}$ for all $i \in \{1, \dots, m'\}$ and $j \in \{1, \dots, m''\}$. This holds, according to the definition of \mathcal{R} : for $(T'', T''') = (T', T')$ by taking $h = 1$ and by observing that pairs $(T'_i, T''_j) \in \mathcal{R}$ because they are added to \mathcal{R} in the case $n = 1$; for any (T'', T''') added to \mathcal{R} in the case n , by taking $h = n + 1$ and by observing that pairs $(T'_i, T''_j) \in \mathcal{R}$ because they are among the pairs that are added to \mathcal{R} in the case $n + 1$. ◀

► **Proposition 16.** *Given a single-out session type T , $\text{reach}(T)$ is finite and it is decidable whether $\text{antOutInf}(T)$.*

Proof. Direct consequence of Lemmas 33, Lemma 37 and the finiteness of antSet_T . ◀

B.4 Proof of Theorem 17

► **Lemma 38.** *Consider two single-out session types T and S . Given a judgement $\Sigma' \vdash T' \leq_t S'$ such that $\emptyset \vdash T \leq_t S \rightarrow^* \Sigma' \vdash T' \leq_t S'$, in such a way that the final rule applied is not RecR_2 , we have that for all $Q \in \text{reach}(S')$ there exist $R \in \text{reach}(S)$ and a sequence of labels γ such that $Q = \text{antOut}(R, \gamma)$.*

Proof. By induction on the length of the sequence of rule applications $\emptyset \vdash T \leq_t S \rightarrow^* \Sigma' \vdash T' \leq_t S'$. In the base case we have $S' = S$. Consider now $Q \in \text{reach}(S')$. Obviously $Q = \text{antOut}(Q, \epsilon)$ with $Q \in \text{reach}(S)$ because $\text{reach}(S) = \text{reach}(S')$.

In the inductive case we proceed by case analysis on the last rule application $\Sigma'' \vdash T'' \leq_t S'' \rightarrow \Sigma' \vdash T' \leq_t S'$. We have two possible cases:

- We can apply the induction hypotheses on the judgement $\Sigma'' \vdash T'' \leq_t S''$. Hence for all $Q'' \in \text{reach}(S'')$ there exist $R \in \text{reach}(S)$ and a sequence of labels γ such that $Q'' = \text{antOut}(R, \gamma)$. Consider now $Q \in \text{reach}(S')$. We proceed by cases on the applied rule.

For the rules In , RecR_1 and Out with $\mathcal{A} = []^1$ we have that $S' \in \text{reach}(S'')$ hence also $Q \in \text{reach}(S'')$ because if $S' \in \text{reach}(S'')$ then $\text{reach}(S') \subseteq \text{reach}(S'')$ by definition of $\text{reach}(_)$.

If the rule is Out with $\mathcal{A} \neq []^1$ we have that $S' = \text{antOut}(R, \gamma \cdot l)$ with $R \in \text{reach}(S)$ and γ such that $S'' = \text{antOut}(R, \gamma)$ and l is the label of the anticipated output. We limit

our analysis to the case in which $Q \notin \text{reach}(S'')$ (in the other cases we can proceed as above). This happens if Q is obtained by applying rule 2. of Definition 15 to remove some but not all the inputs in front of one of the output anticipated in S'' . Consider now the term V corresponding to Q enriched with the anticipated outputs. We have that $V \in \text{reach}(S'')$ hence there exist $R' \in \text{reach}(S)$ and γ' such that $V = \text{antOut}(R', \gamma')$. But $Q = \text{antOut}(R', \gamma' \cdot l)$ hence proving the thesis.

- We cannot apply the induction hypotheses on the judgement $\Sigma'' \vdash T'' \leq_t S''$ because the rule used to obtain $\Sigma'' \vdash T'' \leq_t S''$ is RecR_2 . As RecR_2 cannot be applied in sequence, it is surely possible to apply the induction hypothesis on the previous judgement $\Sigma''' \vdash T''' \leq_t S'''$ such that $\Sigma''' \vdash T''' \leq_t S''' \rightarrow \Sigma'' \vdash T'' \leq_t S''$. Then we have that for all $Q''' \in \text{reach}(S''')$ we have $Q''' = \text{antOut}(R, \gamma)$ with $R \in \text{reach}(S)$ and a sequence of labels γ . We also have that the rule applied in $\Sigma'' \vdash T'' \leq_t S'' \rightarrow \Sigma' \vdash T' \leq_t S'$ is Out , which is the only rule that can be applied after RecR_2 . Let l be the label of the output involved in the application of the Out rule. Consider now $Q \in \text{reach}(S')$. We consider two possible cases:
 - Q is obtained from S' by consuming inputs present in the input context \mathcal{A} used in the last application of the rule Out . Consider now Q''' obtained from S''' by consuming the same inputs and performing the needed unfoldings. Obviously $Q''' \in \text{reach}(S''')$: hence, by induction hypothesis, $Q''' = \text{antOut}(R, \gamma)$ with $R \in \text{reach}(S)$. We have $Q = \text{antOut}(R, \gamma \cdot l)$ hence proving the thesis.
 - Q is obtained from S' by consuming strictly more than a sequence of inputs present in the input context \mathcal{A} used in the last application of the rule Out . This means that $Q \in \text{reach}(W)$ where W is a term starting with an output that populates one of the holes of \mathcal{A} in S'' . But the terms starting with an output that can occur in S'' , assuming $\emptyset \vdash T \leq_t S \rightarrow^* \Sigma'' \vdash T'' \leq_t S''$, are already in $\text{reach}(S)$. In fact the rules do not perform transformations under outputs, excluding those strictly performed by top level unfoldings. Hence $W \in \text{reach}(S)$, which implies $Q \in \text{reach}(S)$ from which the thesis trivially follows (because $Q = \text{antOut}(Q, \epsilon)$). ◀

► **Corollary 39.** *Consider two single-out session types T and S . Given a judgement $\Sigma' \vdash T' \leq_t S'$ such that $\emptyset \vdash T \leq_t S \rightarrow^* \Sigma' \vdash T' \leq_t S'$ and a pair $(T'', S'') \in \Sigma'$, we have that $S'' = \text{antOut}(R, \gamma)$ for some $R \in \text{reach}(S)$ and a sequence of labels γ .*

Proof. Let $(T'', S'') \in \Sigma'$. This pair has been introduced by application of one of the rules RecL , RecR_1 or RecR_2 . But before the application of these rules it is not possible to apply rule RecR_2 , because after such rule only Out can be applied. So the pair (T'', S'') corresponds to a sequence of rule applications $\emptyset \vdash T \leq_t S \rightarrow^* \Sigma'' \vdash T'' \leq_t S''$ in which RecR_2 is not the last applied rule. The thesis directly follows from Lemma 38. ◀

► **Theorem 17.** *Given two single-out session types T and S , the algorithm applied to the initial judgement $\emptyset \vdash T \leq_t S$ terminates.*

Proof. Assume by contraposition that there exists single-out session types T and S such that the algorithm applied to the initial judgement $\emptyset \vdash T \leq_t S$ does not terminate. This means that there exists an infinite sequence of rule applications $\emptyset \vdash T \leq_t S \rightarrow \Sigma_1 \vdash T_1 \leq_t S_1 \rightarrow^* \Sigma_i \vdash T_i \leq_t S_i \rightarrow^*$. Within this infinite sequence, there are infinitely many applications of the unfolding rules RecL , RecR_1 or RecR_2 , that implies the existence of infinitely many distinct pairs (T_j, S_j) that are introduced in the environment (assuming that j ranges over the instances of application of such rules). All these pairs are distinct, otherwise the precedence of the Asmp rule would have blocked the algorithm. It is obvious that the distinct r.h.s. T_j

are finitely many, because every $T_j \in \text{reach}(T)$, which is a finite set. On the contrary, the distinct S_j are infinitely many, but Corollary 39 guarantees that for each of them, there exists $S'_j \in \text{reach}(S)$ and a sequence of labels γ_j such that $S_j = \text{antOut}(S'_j, \gamma_j)$. Within this infinite sequence of pairs (T_j, S_j) d

Due to the finiteness of the possible T_j and S'_j , there exists T'' and S'' such that there exists an infinite subsequence of $(T_{j_1}, S_{j_1}), (T_{j_2}, S_{j_2}), \dots, (T_{j_k}, S_{j_k}), \dots$ such that $T_{j_i} = T''$ and $S_{j_i} = \text{antOut}(S'', \gamma_{j_i})$. It is not restrictive to consider $j_i < j_{i+1}$ for every i . The presence of infinitely many distinct γ_{j_i} for which $\text{antOut}(S'', \gamma_{j_i})$ is defined, guarantees $\text{antOutInf}(S'')$. Moreover, this guarantees also the possibility to define an infinite subsequence $(T_{j_{i_1}}, S_{j_{i_1}}), (T_{j_{i_2}}, S_{j_{i_2}}), \dots, (T_{j_{i_k}}, S_{j_{i_k}}), \dots$ such that $|\gamma_{j_{i_i}}| < |\gamma_{j_{i_{i+1}}}|$. We now consider the leaf sets $\text{leafSet}(S_{j_{i_i}})$. These sets are defined on a finite domain because the subterms of such types starting with a recursive definition or an output, and preceded by inputs only, are taken from $\text{reach}(S)$. This because the algorithm does not apply transformations under recursive definitions or outputs, excluding the effect of the standard top level unfolding of previous recursive definitions, which is considered in the definition of $\text{reach}(S)$. Hence there are only finitely many distinct $\text{leafSet}(S_{j_{i_i}})$, that guarantees the existence of $v < w$ such that $\text{leafSet}(S_{j_{i_v}}) = \text{leafSet}(S_{j_{i_w}})$. Consider now the judgement $\Sigma_{j_{i_w}} \vdash T_{j_{i_w}} \leq_t S_{j_{i_w}}$. We know that $(T_{j_{i_v}}, S_{j_{i_v}}) \in \Sigma_{j_w}$, $T_{j_{i_v}} = T_{j_{i_w}}$, $S_{j_{i_v}} = \text{antOut}(S'', \gamma_{j_{i_v}})$, $S_{j_{i_w}} = \text{antOut}(S'', \gamma_{j_{i_w}})$, $S'' \in \text{reach}(S)$, and $|\gamma_{j_{i_v}}| < |\gamma_{j_{i_w}}|$. Hence it is possible to apply to such judgement the rule Asmp_2 . As Asmp_2 has priority, it should be applied on this judgement thus blocking the sequence of rule applications. But this contradicts the initial assumption of non termination of the algorithm. \blacktriangleleft

B.5 Proof of Theorem 18

► **Definition 40.** Let T', T'' be single-out session types. We say that $T' \text{extAntEq}_T T''$ if there exist $l, \mathcal{A}', \mathcal{A}''$ such that $\text{outUnf}(T') = \mathcal{A}'[\oplus\{l : T'_i\}]^{i \in \{1, \dots, n\}}$ and $\text{outUnf}(T'') = \mathcal{A}''[\oplus\{l : T''_j\}]^{j \in \{1, \dots, m\}}$, with $T'_i \text{antEq}_T T''_j$ for all $i \in \{1, \dots, n\}$ and $j \in \{1, \dots, m\}$.

Moreover, extAntSet_T is the field of extAntEq_T .

Notice that, all terms T'_i , with $i \in \{1, \dots, n\}$ and T''_j , with $j \in \{1, \dots, m\}$, are in $\text{antSet}_T \subseteq \text{reach}(T)$. Moreover, notice that extAntEq_T is obviously an equivalence relation on extAntSet_T .

► **Lemma 41.** Let $T' \in \text{antSet}_T$ and $T'' = \text{antOut}(T', \gamma)$ for some γ . We have that $T'' \in \text{extAntSet}_T$.

Proof. We have to show that there exist l, \mathcal{A} for which we have $\text{outUnf}(\text{antOut}(T', \gamma)) = \mathcal{A}[\oplus\{l : T_i\}]^{i \in \{1, \dots, m\}}$, with $T_i \text{antEq}_T T_j$ for all $i, j \in \{1, \dots, m\}$. We denote $\gamma l = l_{i_1} \dots l_{i_h}$, with $h \geq 1$. For any n , with $1 \leq n \leq h$, considered \mathcal{A}' and terms T_k with $k \in \{1 \dots m_n\}$ such that $\text{outUnf}(\text{antOut}(T', l_{i_1} \dots l_{i_{n-1}})) = \mathcal{A}'[\oplus\{l_{i_n} : T_k\}]^{k \in \{1 \dots m_n\}}$, we have that $\forall i, j \in \{1 \dots m_n\}. T_i \text{antEq}_T T_j$. This is easily shown by induction on n , applying the definition of antEq_T (the base case is directly derived from $T' \text{antEq}_T T'$). The case $n = h$ yields the desired result. \blacktriangleleft

► **Lemma 42.** Let $T', T'' \in \text{extAntSet}_T$ and $\text{leafSet}(T') = \text{leafSet}(T'')$. We have that $T' \text{extAntEq}_T T''$.

Proof. It is easy to see that, from $\text{leafSet}(T') = \text{leafSet}(T'')$, we have $\text{leafSet}(\text{outUnf}(T')) = \text{leafSet}(\text{outUnf}(T''))$. This because $\text{outUnf}()$ causes a leaf T''' belonging to both $\text{leafSet}(T')$

and $\text{leafSet}(T'')$ to yield the same new set of leaves $\text{leafSet}(T''')$ in both T' and T'' . By definition of extAntSet_T we have that exist l', \mathcal{A}' such that $\text{outUnf}(T') = \mathcal{A}'[\oplus\{l' : T'_i\}]^{i \in \{1, \dots, n\}}$, with $T'_i \text{ antEq}_T T'_j$ for all $i, j \in \{1, \dots, n\}$. Similarly, there exist l'', \mathcal{A}'' such that $\text{outUnf}(T'') = \mathcal{A}''[\oplus\{l'' : T''_j\}]^{j \in \{1, \dots, m\}}$, with $T''_i \text{ antEq}_T T''_j$ for all $i, j \in \{1, \dots, m\}$. From the fact that $\text{leafSet}(\text{outUnf}(T')) = \text{leafSet}(\text{outUnf}(T''))$ we have that $l' = l''$ and that: for all T'_i , with $i \in \{1, \dots, n\}$, there exists T''_j , with $j \in \{1, \dots, m\}$, such that $T'_i = T''_j$; and, vice versa, for all T''_j , with $j \in \{1, \dots, m\}$, there exists T'_i , with $i \in \{1, \dots, n\}$, such that $T''_j = T'_i$. Therefore we conclude that $T' \text{ extAntEq}_T T''$. \blacktriangleleft

We now consider a *new subtyping procedure*, we here denote by \leq_s , that is defined exactly as that of \leq_a (defined in Section 3.1 and based on applications of the rules therein over judgments of the form $\Sigma \vdash T \leq_a S$) with the only difference that the **Asmp** rule is removed. Notice that, in the absence of the **Asmp** rule the content of environment Σ is never accessed for reading, so it has no actual effect on the procedure (on rule applications) and can be removed as well, together with updates on such environment made by the rules. As a consequence we will denote \leq_s judgments just by $\vdash T \leq_s S$ for some T and S . Also in this case, we use $\vdash T \leq_s S \rightarrow \vdash T' \leq_s S'$, to denote that the latter can be obtained from the former by one rule application, and $\vdash T \leq_s S \rightarrow_{\text{err}}$, to denote that there is no rule that can be applied to the judgement $\vdash T \leq_s S$.

► **Definition 43.** A *blocking judgment* $\vdash T \leq_s S$, denoted by $\vdash T \leq_s S \rightarrow_{\text{blk}}$, is a judgment such that, for some T', S' we have: $\vdash T \leq_s S \rightarrow^* \vdash T' \leq_s S' \rightarrow_{\text{err}}$ by applying rules **RecL**, **RecR₁** and **RecR₂** only.

► **Definition 44.** An *IO step* a , denoted by $\xrightarrow{a}_{\text{io}}$, with $a \in \{\&l, \oplus_l \mid l \in L\}$ is a sequence of \leq_s rule applications \rightarrow^* such that the last applied rule is an **In** (in the case $a = \&l$, where l is the input label singling out which of the rule premises we consider), or an **Out** rule (in the case $a = \oplus_l$, where l is the output label singling out which of the rule premises we consider) and all other rule applications concern **RecL**, **RecR₁** and **RecR₂** rules only.

► **Definition 45.** $a_1 \dots a_n$, with $n \geq 0$, is a *blocking path* for judgment $\vdash T \leq_s S$ if there exist T', S' such that $\vdash T \leq_s S \xrightarrow{a_1}_{\text{io}} \dots \xrightarrow{a_n}_{\text{io}} \vdash T' \leq_s S' \rightarrow_{\text{blk}}$ (where $T' = T$ and $S' = S$ in the case $n = 0$).

► **Lemma 46.** Let $S \in \text{reach}(Z)$ and $\vdash T \leq_s \text{antOut}(S, \gamma)$, $\vdash T \leq_s \text{antOut}(S, \beta)$ be such that: $|\gamma| < |\beta|$ and $\text{antOut}(S, \beta) \text{ extAntEq}_Z \text{antOut}(S, \gamma)$. If $a_1 \dots a_n$, with $n \geq 0$, is a blocking path for $\vdash T \leq_s \text{antOut}(S, \beta)$ then there exists a m long prefix of $a_1 \dots a_n$, with $0 \leq m \leq n$, that is a blocking path for $\vdash T \leq_s \text{antOut}(S, \gamma)$.

Proof. The proof is by induction on $n \geq 0$.

We start by proving the base case $n = 0$. That is $\vdash T \leq_s \text{antOut}(S, \gamma) \rightarrow_{\text{blk}}$, i.e. for some T', S' we have: $\vdash T \leq_s \text{antOut}(S, \gamma) \rightarrow^* \vdash T' \leq_s S' \rightarrow_{\text{err}}$ by applying rules **RecL**, **RecR₁** and **RecR₂** only.

We first observe that $\vdash T \leq_s \text{antOut}(S, \gamma) \xrightarrow{a}_{\text{io}}$ is not possible for any $a \in \{\&l, \oplus_l \mid l \in L\}$. This because: if we had $\vdash T \leq_s \text{antOut}(S, \gamma) \xrightarrow{\&l}_{\text{io}}$ for some $l \in L$, then $\text{antOut}(S, \beta) = \&\{l_i : T_i\}_{i \in I}$ with $l = l_i$ for some $i \in I$, hence we would have that also $\vdash T \leq_s \text{antOut}(S, \beta) \xrightarrow{\&l}_{\text{io}}$; and if we had $\vdash T \leq_s \text{antOut}(S, \gamma) \xrightarrow{\oplus_l}_{\text{io}}$ for some $l \in L$, then, since $\text{antOut}(S, \beta) \text{ extAntEq}_Z \text{antOut}(S, \gamma)$, we would have that also $\vdash T \leq_s \text{antOut}(S, \beta) \xrightarrow{\oplus_l}_{\text{io}}$.

Therefore, given that it is not possible that $\vdash T \leq_s \text{antOut}(S, \gamma) \rightarrow^* \vdash \text{end} \leq_s \text{end}$ by applying rules **RecL**, **RecR₁** and **RecR₂** only (because otherwise $\text{antOut}(S, \beta)$ would not be defined), we conclude $\vdash T \leq_s \text{antOut}(S, \gamma) \rightarrow_{\text{blk}}$ (notice that the number of times a **RecL**,

RecR_1 or RecR_2 is applicable to a judgment is finite because we do not have unguarded recursion and RecR_2 cannot be consecutively applied for more than one time).

We now consider the induction case for blocking path $a_1 \dots a_n$ of length $n \geq 1$.

We first consider the case $a_1 = \&l$ for some $l \in L$. Given that $\text{antOut}(S, \beta)$ is defined and that $\vdash T \leq_s \text{antOut}(S, \beta) \xrightarrow{\&l}_{\text{io}}$, we deduce that $\text{antOut}(S, \gamma)$ is: either $\oplus\{l' : T'\}$ (possibly preceded by some recursion operators), for some l', T' ; or $\&\{l_i : T_i\}_{i \in I}$ (possibly preceded by some recursion operators), for some terms T_i and labels l_i such that $l = l_i$ for some $i \in I$. In the first case we have $\vdash T \leq_s \text{antOut}(S, \gamma) \rightarrow_{\text{blk}}$, hence the lemma trivially holds; in the second case we have $\vdash T \leq_s \text{antOut}(S, \gamma) \xrightarrow{\&l}_{\text{io}}$ and we proceed with the proof. We have that there exist T', S', σ such that $\vdash T \leq_s \text{antOut}(S, \gamma) \xrightarrow{\&l}_{\text{io}} T' \leq_s \text{antOut}(S', \gamma')$ and $\vdash T \leq_s \text{antOut}(S, \beta) \xrightarrow{\&l}_{\text{io}} T' \leq_s \text{antOut}(S', \beta')$, with $\gamma = \sigma\gamma'$ and $\beta = \sigma\beta'$. In particular S' is obtained from S by removing all its initial (single-)outputs (and intertwined recursions, that are unfolded) until the first input $\&\{l_i : T_i\}_{i \in I}$ is reached, which is also removed, thus yielding $S' = T_i$ for the $i \in I$ such that $l = l_i$. This corresponds, in the definition of $\text{reach}(Z)$ (Definition 15), to repeatedly applying, starting from $S \in \text{reach}(Z)$, rules 3 and 4 and finally rule 2, thus yielding $S' \in \text{reach}(Z)$. Notice that σ is the sequence of labels of the initial outputs that were removed during this procedure and that, obviously, $|\gamma'| < |\beta'|$.

Now, in order to be able to apply the induction hypothesis we have also to show that $\text{antOut}(S, \beta') \text{extAntEq}_Z \text{antOut}(S, \gamma')$. We observe that $\text{antOut}(S, \gamma') \text{extAntEq}_Z \text{antOut}(S, \gamma)$. This holds because $\text{antOut}(S, \gamma)$ is a $\&\{l_i : T_i\}_{i \in I}$ term, with $l = l_i$ for some $i \in I$, possibly preceded by some recursion operators, and from the following observations: obviously, for any t, T'' , it holds $\mu t. T'' \text{extAntEq}_Z T'' \{\mu t. T'' / t\}$; and $\text{leafSet}(T_i) \subseteq \text{leafSet}(\&\{l_i : T_i\}_{i \in I})$. In the same way, we have $\text{antOut}(S, \beta') \text{extAntEq}_Z \text{antOut}(S, \beta)$.

It is therefore possible to apply the induction hypothesis to $T' \leq_s \text{antOut}(S', \gamma')$ and $T' \leq_s \text{antOut}(S', \beta')$ that possesses the shorter blocking path $a_2 \dots a_n$.

Finally, we consider the case $a_1 = \oplus l$ for some $l \in L$. Since $\vdash T \leq_s \text{antOut}(S, \beta) \xrightarrow{\oplus l}_{\text{io}}$ and $\text{antOut}(S, \beta) \text{extAntEq}_Z \text{antOut}(S, \gamma)$, we have that also $\vdash T \leq_s \text{antOut}(S, \gamma) \xrightarrow{\oplus l}_{\text{io}}$. In particular, we have that there exists T' such that $\vdash T \leq_s \text{antOut}(S, \gamma) \xrightarrow{\oplus l}_{\text{io}} T' \leq_s \text{antOut}(S, \gamma l)$ and $\vdash T \leq_s \text{antOut}(S, \beta) \xrightarrow{\oplus l}_{\text{io}} T' \leq_s \text{antOut}(S, \beta l)$, where, obviously, $|\gamma l| < |\beta l|$. Moreover, since $\text{antOut}(S, \beta) \text{extAntEq}_Z \text{antOut}(S, \gamma)$ it is immediate to show (by applying the definitions of antOut , extAntEq and antEq) that also $\text{antOut}(S, \beta l) \text{extAntEq}_Z \text{antOut}(S, \gamma l)$.

It is therefore possible to apply the induction hypothesis to $T' \leq_s \text{antOut}(S, \gamma l)$ and $T' \leq_s \text{antOut}(S, \beta l)$ that possesses the shorter blocking path $a_2 \dots a_n$. \blacktriangleleft

► **Theorem 18.** Given two single-out session types T and S , we have that there exist Σ', T', S' such that $\emptyset \vdash T \leq_a S \rightarrow^* \Sigma' \vdash T' \leq_a S' \rightarrow_{\text{err}}$ if and only if there exist Σ'', T'', S'' such that $\emptyset \vdash T \leq_t S \rightarrow^* \Sigma'' \vdash T'' \leq_t S'' \rightarrow_{\text{err}}$.

Proof. We consider the two implications separately starting from the *if* part. Assume that $\emptyset \vdash T \leq_t S \rightarrow^* \Sigma'' \vdash T'' \leq_t S'' \rightarrow_{\text{err}}$. In this sequence of rule applications, the new rule Asmp2 is never used otherwise the sequence would terminate successfully by applying such a rule. Hence, by applying the same sequence of rules, we have $\emptyset \vdash T \leq_a S \rightarrow^* \Sigma' \vdash T' \leq_a S'$ with $T'' = T'$, $S'' = S'$ and $\Sigma'' = \Sigma'$. We have that $\Sigma' \vdash T' \leq_a S' \rightarrow_{\text{err}}$, otherwise if a rule could be applied to this judgement, the same rule could be applied also to $\Sigma'' \vdash T'' \leq_t S''$ thus contradicting the assumption $\Sigma'' \vdash T'' \leq_t S'' \rightarrow_{\text{err}}$.

We now move to the *only if* part. Assume that $\emptyset \vdash T \leq_a S \rightarrow^* \Sigma' \vdash T' \leq_a S' \rightarrow_{\text{err}}$ and that, by contradiction, $\emptyset \vdash T \leq_t S \rightarrow^* \Sigma'' \vdash T'' \leq_t S'' \rightarrow_{\text{err}}$ does not hold.

From $\emptyset \vdash T \leq_a S \rightarrow^* \Sigma' \vdash T' \leq_a S' \rightarrow_{\text{err}}$ (since in this sequence of rule applications the **Asmp** rule is never used, otherwise the sequence would terminate successfully by applying such a rule), by applying the same sequence of rules, we have $\vdash T \leq_s S \rightarrow^* \vdash T' \leq_s S'$.

We now observe that, since we assumed (by contradiction) that we do not get the error when using the \leq_t procedure, there must exist at least a triple Σ''', T''', S''' such that: $\emptyset \vdash T \leq_t S \rightarrow^* \Sigma''' \vdash T''' \leq_t S'''$ (and correspondingly $\vdash T \leq_s S \rightarrow^* \vdash T''' \leq_s S'''$ because the **Asmp** and **Asmp2** rules, that would have led to successful termination, cannot have been applied), $\Sigma''' \vdash T''' \leq_t S'''$ successfully terminates by applying the **Asmp** or **Asmp2** rule, and $\vdash T''' \leq_s S'''$ has a blocking path.

Let us now consider one of such triples Σ''', T''', S''' (possessing the above stated properties) that has a blocking path of minimal length, i.e. there is no other Σ''', T''', S''' triple of the kind above such that $\vdash T''' \leq_s S'''$ has a shorter blocking path. Let $a_1 \dots a_n$ be such a path. Since the **Asmp** or **Asmp2** rule is applied to $\Sigma''' \vdash T''' \leq_t S'''$, we have $S''' = \text{antOut}(S, \beta)$ (in the case of **Asmp** this is obtained by Corollary 39).

We now consider γ such that $(T''', \text{antOut}(S, \gamma)) \in \Sigma'''$ was used in the premise of **Asmp** or **Asmp2** rule: $\gamma = \beta$ in the case of the **Asmp** rule, $|\gamma| < |\beta|$ in the case of the **Asmp2** rule. Moreover, let us also consider Σ_γ to be the environment such that $\emptyset \vdash T \leq_t S \rightarrow^* \Sigma_\gamma \vdash T''' \leq_t \text{antOut}(S, \gamma)$, where $\Sigma_\gamma \vdash T''' \leq_t \text{antOut}(S, \gamma)$ is the judgment to which the rule the caused $(T''', \text{antOut}(S, \gamma))$ to be inserted in the environment was applied.

We now observe that there exists a m long prefix of $a_1 \dots a_n$, with $0 \leq m \leq n$, that is a blocking path for $\vdash T''' \leq_s \text{antOut}(S, \gamma)$. This is obvious in the case $\gamma = \beta$; it is due to Lemma 46 in the case $|\gamma| < |\beta|$: we obtain $\text{antOut}(S, \beta) \text{extAntEq}_Z \text{antOut}(S, \gamma)$ as needed by such a Lemma from the statements in the premise of rule **Asmp2** and by applying Lemmas 37, 41 and 42.

Since we assumed (by contradiction) that $\emptyset \vdash T \leq_t S \rightarrow^* \Sigma'' \vdash T'' \leq_t S'' \rightarrow_{\text{err}}$ does not hold, this would be possible only if there existed a triple $\Sigma''''', T''''', S'''''$ such that: there is a sequence of rule applications $\Sigma_\gamma \vdash T''' \leq_t \text{antOut}(S, \gamma) \rightarrow^* \Sigma'''' \vdash T'''' \leq_t S''''$ that is a prefix of the sequence of rule applications of the blocking path for $\vdash T''' \leq_s \text{antOut}(S, \gamma)$; and $\Sigma'''' \vdash T'''' \leq_t S''''$ successfully terminates by applying the **Asmp** or **Asmp2** rule. Notice that such a sequence $\Sigma_\gamma \vdash T''' \leq_t \text{antOut}(S, \gamma) \rightarrow^* \Sigma'''' \vdash T'''' \leq_t S''''$ should necessarily include the application of, at least, an **In** rule (causing the algorithm to branch), because otherwise (given that $\Sigma_\gamma \vdash T'''' \leq_t \text{antOut}(S, \gamma) \rightarrow^* \Sigma'''' \vdash T'''' \leq_t \text{antOut}(S, \beta)$) we could not have that $\Sigma'''' \vdash T'''' \leq_t S''''$ successfully terminates by applying the **Asmp** or **Asmp2** rule.

However the existence of such a triple $\Sigma''''', T''''', S'''''$ is not possible, because $\vdash T'''' \leq_s S''''$ would have a k long blocking path with $k < n$ (being such a path strictly shorter than that of $\vdash T''' \leq_s \text{antOut}(S, \gamma)$), thus violating the minimality assumption about the blocking path length of the Σ''', T''', S''' triple. \blacktriangleleft

C Proofs of Section 4

C.1 Proof of Theorem 22

► **Definition 47.** Let $M = (\{q_1, \dots, q_n\}, \Sigma, \Gamma, \$, s, \delta)$ and let $\#$ be a special character not in Γ . We denote with $\llbracket M \rrbracket$ the following single-consuming queue machine $(\{q_1, \dots, q_n, q'_1, \dots, q'_n\}, \Sigma, \Gamma \cup \{\#\}, \$, s, \delta')$ with δ' defined as follows:

- $\delta'(q_i, a) = (q'_j, \epsilon)$ if $\delta(q_i, a) = (q_j, \epsilon)$
- $\delta'(q_i, a) = (q_j, \gamma)$ if $\delta(q_i, a) = (q_j, \gamma)$ with $\gamma \neq \epsilon$
- $\delta'(q_i, \#) = (q'_i, \epsilon)$
- $\delta'(q'_i, a) = (q_j, \#)$ if $\delta(q_i, a) = (q_j, \epsilon)$

- $\delta'(q'_i, a) = (q_j, \gamma)$ if $\delta(q_i, a) = (q_j, \gamma)$ with $\gamma \neq \epsilon$
- $\delta'(q'_i, \#) = (q_i, \#)$

Given a configuration (q, γ) of $\llbracket M \rrbracket$, we denote with $\llbracket (q, \gamma) \rrbracket$ the configuration (z, β) where $z = q$, if $q \in \{q_1, \dots, q_n\}$, or $z = q_i$, if $q = q'_i$, while β is obtained from γ by removing each instance of the special symbol $\#$.

► **Lemma 48.** *Let $M = (Q, \Sigma, \Gamma, \$, s, \delta)$ be a queue machine and let $x \in \Sigma^*$. If $(s, x\$) \rightarrow_M^* (q, \gamma)$ then there exists a configuration (q', γ') such that $(s, x\$) \rightarrow_{\llbracket M \rrbracket}^* (q', \gamma')$ with $\llbracket (q', \gamma') \rrbracket = (q, \gamma)$.*

Proof. By induction on the number of steps in the sequence $(s, x\$) \rightarrow_M^* (q, \gamma)$. The base case is trivial. In the inductive case we perform a case analysis. The unique non trivial case is when the configuration reached by $\llbracket M \rrbracket$ according to the inductive hypothesis has the queue starting with the special symbol $\#$. In this case, $\llbracket M \rrbracket$ must perform more transitions, first to consume all the instances of $\#$ in front of the queue and then to mimick the new transition of M . ◀

► **Lemma 49.** *Let $M = (Q, \Sigma, \Gamma, \$, s, \delta)$ be a queue machine and let $x \in \Sigma^*$. If $(s, x\$) \rightarrow_{\llbracket M \rrbracket}^* (q, \gamma)$ then $(s, x\$) \rightarrow_M^* \llbracket (q, \gamma) \rrbracket$.*

Proof. By induction on the number of steps in the sequence $(s, x\$) \rightarrow_{\llbracket M \rrbracket}^* (q, \gamma)$. The base case is trivial. In the inductive case we perform a case analysis. The unique non trivial case is when γ starts with the special symbol $\#$. In this case, M does not perform any new transition as if (q', γ') is the new configuration we have that $\llbracket (q, \gamma) \rrbracket = \llbracket (q', \gamma') \rrbracket$. ◀

► **Theorem 22.** *Given a single consuming queue machine M and an input x , the termination of M on x is undecidable.*

Proof. The thesis directly follows from the Turing completeness of queue machines, and the two above Lemmas that guarantee that given a queue machine M and an input x , s terminates on x if and only if the single-consuming queue machine $\llbracket M \rrbracket$ terminates on x . This is guaranteed by the fact that if $\llbracket M \rrbracket$ reaches a configuration with the queue containing only instances of $\#$, it is guaranteed to eventually terminate by emptying the queue. ◀

C.2 Proof of Theorem 25

Notation. Given a sequence of queue symbols γ , we denote with $\llbracket \gamma \rrbracket_u$ the set of session types that can be obtained from $\llbracket \gamma \rrbracket$ by replacing each subterm $T_k = T''$ (considering the term T'' as defined in Figure 2) with $\text{antOut}(T'', l_{i_1^k} \dots l_{i_{n_k}^k})$, for sequences of labels $l_{i_1^k} \dots l_{i_{n_k}^k}$ with $n_k \geq 0$. Observe that $\llbracket \gamma \rrbracket_u$ is well defined because T'' can anticipate every possible sequence of outputs.

► **Lemma 50.** *Given a single-consuming queue machine $M = (Q, \Sigma, \Gamma, \$, s, \delta)$ and an input string $x \in \Sigma^*$, if $\llbracket s \rrbracket^\emptyset \leq \llbracket x\$ \rrbracket$ then M does not terminate on x .*

Proof. We need a preliminary result: given $(q, \gamma) \rightarrow_M (q', \gamma')$ and a term $S \in \llbracket \gamma \rrbracket_u$, if $\llbracket q \rrbracket^\emptyset \leq S$ then there exists $S' \in \llbracket \gamma' \rrbracket_u$ such that $\llbracket q' \rrbracket^\emptyset \leq S'$. In fact, assuming $\gamma = C_1 \dots C_m$ and $\delta(q, C_1) = (q', B_1^{C_1} \dots B_{n_{C_1}}^{C_1})$, we have $\gamma' = C_2 \dots C_m B_1^{C_1} \dots B_{n_{C_1}}^{C_1}$. Consider now $S \in \llbracket \gamma \rrbracket_u$. Having $\llbracket q \rrbracket^\emptyset \leq S$, by one application of item 4. of Definition 4, one application of item 3., and n_A applications of item 2., we can conclude that there exists $S' \in \llbracket \gamma' \rrbracket_u$ such that $\llbracket q' \rrbracket^\emptyset \leq S'$.

We now prove the thesis by showing that assuming that M accepts x we have $\llbracket s \rrbracket^\emptyset \not\leq \llbracket x\$ \rrbracket$. By definition of queue machines, we have that: M accepts x implies $(s, x\$) \rightarrow_M^* (q, \epsilon)$. Assume

now, by contraposition, that $\llbracket s \rrbracket^\emptyset \leq \llbracket x\$ \rrbracket$. As $(s, x\$) \rightarrow_M^* (q, \epsilon)$, by repeated application of the above preliminary result we have that exists $S' \in \llbracket \epsilon \rrbracket_u$ such that $\llbracket q \rrbracket^\emptyset \leq S'$. But this cannot hold because $\llbracket q \rrbracket^\emptyset$ is a recursive definition that upon unfolding begins with an input that implies (according to items 4. and 3. of Definition 4) that also S' (once unfolded) starts with an input. But this is false, in that, by definition of the queue encoding $\llbracket \epsilon \rrbracket = \mu t \oplus \left\{ A : \& \left(\{ A : t \} \uplus \{ A' : T'' \}_{A' \in \Gamma \setminus \{A\}} \right) \right\}_{A \in \Gamma}$. \blacktriangleleft

► **Lemma 51.** *Given a single-consuming queue machine $M = (Q, \Sigma, \Gamma, \$, s, \delta)$ and an input string $x \in \Sigma^*$, if M does not terminate on x then $\llbracket s \rrbracket^\emptyset \leq \llbracket x\$ \rrbracket$.*

Proof. Assuming that M does not accept x we show that $\llbracket s \rrbracket^\emptyset \leq \llbracket x\$ \rrbracket$. When a queue machine does not accept an input, the corresponding computation never ends. In our case, this means that there is an infinite sequence $(s, x\$) = (q_0, \gamma_0) \rightarrow_M (q_1, \gamma_1) \rightarrow_M \cdots \rightarrow_M (q_i, \gamma_i) \rightarrow_M \cdots$. Let \mathcal{C} be the set of reachable configurations, i.e. $\mathcal{C} = \{(q_i, \gamma_i) \mid i \geq 0\}$. We now define a relation \mathcal{R} on types, where T' and T'' are as in Figure 2, $T_0 = \oplus \{A : T''\}_{A \in \Gamma}$ and $T_n = \& \{A : T_{n-1}\}_{A \in \Gamma}$:

$$\begin{aligned}
\mathcal{R} = & \{ \quad (\llbracket q \rrbracket^\emptyset, S_{C_1 \dots C_m}), (\& \{ A : \llbracket B_1^A \dots B_{n_A}^A \rrbracket_{q'}^\emptyset \}_{A \in \Gamma}, S_{C_1 \dots C_m}), \\
& (\llbracket B_1^{C_1} \dots B_{n_{C_1}}^{C_1} \rrbracket_{q'}^\emptyset, S_{C_2 \dots C_m}), (\llbracket B_2^{C_1} \dots B_{n_{C_1}}^{C_1} \rrbracket_{q'}^\emptyset, S_{C_2 \dots C_m B_1^{C_1}}), \\
& \dots \\
& (\llbracket B_{n_{C_1}}^{C_1} \rrbracket_{q'}^\emptyset, S_{C_2 \dots C_m B_1^{C_1} \dots B_{n_{C_1}-1}^{C_1}}) \\
& \mid (q, C_1 \dots C_m) \in \mathcal{C}, \delta(q, C_1) = (q', B_1^{C_1} \dots B_{n_{C_1}}^{C_1}), S_\gamma \in \llbracket \gamma \rrbracket_u \} \\
\cup & \{ \quad (\llbracket q \rrbracket^\emptyset, T_n), (\& \{ A : \llbracket B_1^A \dots B_{n_A}^A \rrbracket_{q'}^\emptyset \}_{A \in \Gamma}, T_n), \\
& (\llbracket B_1^{C_1} \dots B_{n_{C_1}}^{C_1} \rrbracket_{q'}^\emptyset, T_m), (\llbracket B_2^{C_1} \dots B_{n_{C_1}}^{C_1} \rrbracket_{q'}^\emptyset, T_m), \\
& \dots \\
& (\llbracket B_{n_{C_1}}^{C_1} \rrbracket_{q'}^\emptyset, T_m) \\
& \mid (q, C_1 \dots C_m) \in \mathcal{C}, \delta(q, C_1) = (q', B_1^{C_1} \dots B_{n_{C_1}}^{C_1}), \\
& \text{if } \exists q'', C \text{ s.t. } \delta(q, C) = (q'', \epsilon) \text{ then } n \geq 2 \text{ else } n \geq 1, m \geq 0 \} \\
\cup & \{ \quad (T', T_n), (\& \{ A_1 : \oplus \{ A_2 : T' \}_{A_2 \in \Gamma} \}_{A_1 \in \Gamma}, T_n), (\oplus \{ A_2 : T' \}_{A_2 \in \Gamma}, T_m) \\
& \mid n \geq 1, m \geq 0 \} \\
\cup & \{ \quad (T', S_\gamma), (\& \{ A_1 : \oplus \{ A_2 : T' \}_{A_2 \in \Gamma} \}_{A_1 \in \Gamma}, S_\gamma), (\oplus \{ A_2 : T' \}_{A_2 \in \Gamma}, S_\gamma) \\
& \mid \gamma \in \Gamma^*, S_\gamma \in \llbracket \gamma \rrbracket_u \}
\end{aligned}$$

We have that the above \mathcal{R} is an asynchronous subtyping relation because each of the pairs satisfies the conditions in Definition 4 thanks to the presence of other pairs in \mathcal{R} . We can conclude observing that $(s, x\$) \in \mathcal{C}$ implies that $(\llbracket q \rrbracket^\emptyset, \llbracket x\$ \rrbracket)$ belongs to the above asynchronous subtyping relation \mathcal{R} , hence $\llbracket q \rrbracket^\emptyset \leq \llbracket x\$ \rrbracket$. \blacktriangleleft

► **Theorem 25.** *Given a single consuming queue machine $M = (Q, \Sigma, \Gamma, \$, s, \delta)$ and an input string $x \in \Sigma^*$, we have $\llbracket s \rrbracket^\emptyset \leq \llbracket x\$ \rrbracket$ if and only if M does not terminate on x .*

Proof. Direct consequence of Lemmas 50 and 51. \blacktriangleleft

C.3 Proof of Theorem 30

► **Lemma 28.** *Given a queue machine M and an input x , it is undecidable whether M terminates and is bound on x .*

Proof. We first prove that boundedness is undecidable. If, by contraposition, boundedness was decidable, termination could be decided by first checking boundedness, and then perform a finite state analysis of the queue machine behaviour. More precisely, termination on bounded queue machines can be decided by forward exploration of the reachable configurations until a terminating configuration is found, or a cycle is detected by reaching an already visited configuration.

We now conclude by observing that given a queue machine M and the input x , it is not possible to decide whether M *does not terminate and is bound on x* . Assume by contraposition one could decide the above property of queue machines. Then boundedness could be decided as follows: transform M in a new machine M' that behaves like M plus an additional special symbol $\#$ which is enqueued every time it is dequeued; boundedness of M on input x can be decided by checking the above property on M' and input $\#x$ (in fact M' never terminates and is bound on $\#x$ if and only if M is bound on x). ◀

► **Theorem 30.** *Given a single consuming queue machine $M = (Q, \Sigma, \Gamma, \$, s, \delta)$ and an input string x , we have that $\llbracket s \rrbracket^0 \leq_b \llbracket x\$ \rrbracket$ if and only if M does not terminate and is bound on x .*

Proof. We need a preliminary result: given $(q, \gamma) \rightarrow_M (q', \gamma')$, if $\llbracket q \rrbracket^0 \leq \llbracket \gamma \rrbracket$ then we also have that $\llbracket q' \rrbracket^0 \leq \llbracket \gamma' \rrbracket$. In fact, assuming $\gamma = C_1 \cdots C_m$ and $\delta(q, C_1) = (q', B_1^{C_1} \cdots B_{n_{C_1}}^{C_1})$, we have $\gamma' = C_2 \cdots C_m B_1^{C_1} \cdots B_{n_{C_1}}^{C_1}$. Having $\llbracket q \rrbracket^0 \leq \llbracket \gamma \rrbracket$, by one application of item 4. of Definition 4, one application of item 3., and n_A applications of item 2., we can conclude that $\llbracket q' \rrbracket^0 \leq \llbracket \gamma' \rrbracket$.

We now observe that if M is not bound on x we have that it is not possible to have $\llbracket s \rrbracket^0 \leq_b \llbracket x\$ \rrbracket$. Assume by contraposition that $\llbracket s \rrbracket^0 \leq_b \llbracket x\$ \rrbracket$. From the previous preliminary result, we have that also $\llbracket q' \rrbracket^0 \leq_b \llbracket \gamma' \rrbracket$ for each reachable configuration (q', γ') . But due to unboundedness of M on x there is no limit to the length of γ' . This means that the session types $\llbracket \gamma' \rrbracket$ have input contexts of unbounded length. But as M is single-consuming, we have that $\llbracket q' \rrbracket^0$ has at most two inputs before the first output. Hence, in order to relate $\llbracket q' \rrbracket^0$ and $\llbracket \gamma' \rrbracket$, we need a relation that contains pairs with the l.h.s. starting with an output and the r.h.s. with an input context of unbounded depth. But this cannot hold if we fix a maximal depth k to the depth of the input context.

Now we observe that $\llbracket s \rrbracket^0 \leq_b \llbracket x\$ \rrbracket$ if and only if M does not terminate and is bound on x . If $\llbracket s \rrbracket^0 \leq_b \llbracket x\$ \rrbracket$ then we also have that $\llbracket s \rrbracket^0 \leq \llbracket x\$ \rrbracket$ that implies, for Lemma 50, that M does not terminate on input x . We also have that M is bound on x in the light of the previous observation. Consider now that M does not terminate. As in the proof of Lemma 51, we define $\mathcal{C} = \{(q_i, \gamma_i) \mid (s, x\$) = (q_0, \gamma_0) \rightarrow_M (q_1, \gamma_1) \rightarrow_M \cdots \rightarrow_M (q_i, \gamma_i), i \geq 0\}$ and the following relation \mathcal{R} on types (which is a simplified version w.r.t. the relation in that

proof):

$$\begin{aligned}
 \mathcal{R} = & \{ \quad (\llbracket q \rrbracket^\emptyset, \llbracket C_1 \cdots C_m \rrbracket), (\&\{A: \{B_1^A \cdots B_{n_A}^A\}_{q'}^\emptyset\}_{A \in \Gamma}, \llbracket C_1 \cdots C_m \rrbracket), \\
 & (\{B_1^{C_1} \cdots B_{n_{C_1}}^{C_1}\}_{q'}^\emptyset, \llbracket C_2 \cdots C_m \rrbracket), (\{B_2^{C_1} \cdots B_{n_{C_1}}^{C_1}\}_{q'}^\emptyset, \llbracket C_2 \cdots C_m B_1^{C_1} \rrbracket), \\
 & \dots \\
 & (\{B_{n_{C_1}}^{C_1}\}_{q'}^\emptyset, \llbracket C_2 \cdots C_m B_1^{C_1} \cdots B_{n_{C_1-1}}^{C_1} \rrbracket) \\
 & | \quad (q, C_1 \cdots C_m) \in \mathcal{C}, \delta(q, C_1) = (q', B_1^{C_1} \cdots B_{n_{C_1}}^{C_1}) \}
 \end{aligned}$$

Remember that in the terms $\llbracket _ \rrbracket_q^S$ the alternative semantics for the \uplus operator is considered: $\{l_i : T_i\}_{i \in I} \uplus \{l_j : T_j\}_{j \in J} = \{l_i : T_i\}_{i \in I}$. This relation is an asynchronous subtyping relation, and moreover boundedness of M on x guarantees boundedness on the length of the reachable queue contents $C_1 \cdots C_m$, that implies boundedness of the depth of the input contexts of the r.h.s. of all the pairs in \mathcal{R} . This proves the thesis, i.e. $\llbracket s \rrbracket^\emptyset \leq_b \llbracket x \$ \rrbracket$. \blacktriangleleft